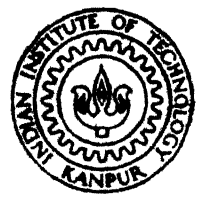
A NATURAL LANGUAGE INTERFACE TO RELATIONAL DATABASES

By

V. V. R. S. S. N. R. KUMAR



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

FEBRUARY, 1985

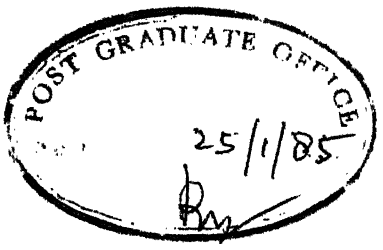
A NATURAL LANGUAGE INTERFACE TO RELATIONAL DATABASES

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
V. V. R. S. S. N. R. KUMAR

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
FEBRUARY, 1985

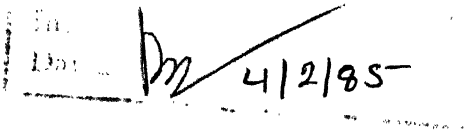


C E R T I F I C A T E

This is to certify that the thesis entitled 'A NATURAL LANGUAGE INTERFACE TO RELATIONAL DATABASES' is a report of work carried out under my supervision by V.V.R.S.S.N.R. Kumar and that it has not been submitted elsewhere for a degree.

Dr. Rajeev Sangal
Assistant Professor
Department of Computer Science
and Engineering
Indian Institute Of Technology, Kanpur

Kanpur
January, 1985



ACKNOWLEDGEMENTS

I take this opportunity to thank my project guide Dr. Rajeev Sangal for taking close interest in my work and giving me invaluable suggestions and references. To him goes the credit of introducing to me the new areas like natural language processing in particular and artificial intelligence in general. I owe him a lot for the long hours he spent in formalising the ideas, implementing the algorithms and more importantly in improving my writing-up capabilities.

I thank Dr. H. Karnick for giving me his time readily whenever I asked him.

I wish to take this opportunity to thank Y.V.Srinivas for strengthening me during the psychological ebbs I suffered. His words of moral support during the periods of my breakdown can never leave my memory.

I convey my thanks to Vizzu for making my stay at Kanpur a memorable one and for his help in preparing the thesis.

Finally, my thanks to all my friends who made those DEC-10 nights happy.

C O N T E N T S

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

When two or more agents cooperate to perform a task, their communicating ability decides their success. Communication is equally important whether the agents are all people, or all programs or some of each.

Since the inception of computers as information systems, trials have been made to improve communication between programs that retrieve information from large amounts of data, and the people who need this information. Man-machine communication using language caters to broader needs than other schemes like pointing, menu selection scheme, drawing, etc.

Natural Language Interface to a database is a program that accepts queries in a Natural Language (like English) and generates a formal query for accessing the given database. A query processor processes the formal query and generates the response which is presented to the user.

A typical example is in management information systems where a user wishes to extract information from a database to make decisions. Ideally, he wishes to interact in his own terminology, rather than consult a programmer. To help such a user, we need a Natural Language Interface to database. Another example is when a casual user wishes to find information from a database. Examples of this situation are:
a reader in a library, a visitor to a firm.

Most of the existing NLI systems are very finely tuned to the specific domain of their discourse. Porting them to a new database is as difficult as redesigning the system for a new database. This motivated us to design a system that comprehends the general principles of Natural Language Interface to databases and eases the portability to new databases. We aimed at designing a Natural Language Interface system that takes database specification as a parameter. To transport the system to a new database, only the new parameters have to be supplied.

1.2 OBJECTIVE

The objective is to build a system that accepts a large subset of English queries on its domain of discourse and convert them into formal queries. At the same time,the

system should be easily portable to new database . There are four aspects which clearly define the capabilities of a Natural Language Interface. They are, the potential users of the system, the portability of the system to a new database, the language accepted, and the constraints on the database structure. We shall formulate our objectives in terms of these four aspects.

1.2.1 USER

The system is aimed at a novice or casual user. He is not expected to know the organisation of the database. This constrains the language accepted to be close to English rather than a command language.

1.2.2 PORTABILITY

The system is aimed to be independent of any particular database.

When the system is to be ported to a new database, the designer specifies the parameters of the new database. We propose to formalise this database specification and algorithmise the translation of these specifications into system parameters.

1.2.3 LANGUAGE

By Natural Language Interface we rarely mean the entire syntax of the Natural Language. Instead, we propose to design the system for a reasonably large subset of Natural Language. Also, the system accepts minor errors in syntax(like disagreement in "case" and "number"). The language should enable the user to learn quickly what types of constructs are accepted and what others are not.

The system does not attempt to understand the query nor to capture the full semantics of the query. Instead, the knowledge associated with certain database specific words of the query and syntactic structure of the query contribute to the conversion from the English to Formal query. Although the system does not understand the query, we claim that the syntactic regularity and the database specific words together determine the formal query almost unambiguously.

1.2.4 DATABASE

We assume that a relational database with all its relations, fields, and the keys is given. We propose to design the system for any relational database unlike some systems (like LUNAR see section 2.2.3) which assume a flat file as database.

1.3 ORGANISATION OF THE THESIS

Chapter 2 gives a brief survey of Natural Language Interface systems and points out their drawbacks viewed in the light of our objectives. It also introduces the basic formalisms of Natural Language parsers. Chapter 3 gives an overview of our system which has four phases. Chapters 4,5,6 discuss these phases in detail and chapter 7 contains the conclusions.

CHAPTER 2

SURVEY OF RELATED SYSTEMS AND ATNS AS PARSER

2.1 INTRODUCTION

A brief survey of relevant natural language systems, followed by a brief introduction to Augmented Transition Networks is the subject of this chapter

2.2 NATURAL LANGUAGE SYSTEMS

The Natural Language(henceforth NL) Interfaces discussed here are PLANES, RANDEZVOUS, LUNAR and LADDER. Each of the systems is desisned for a specific data base. We examine, in brief, how each system works and identify the basic principles of each system.

We shall appraise each system by its adaptability to a new data base; this is achieved by examining the sensititvity of the principles adopted in each system to a change in the specific data base for which it is desisned. The second aspect for appraisal is the habitability of the language accepted by each. These parameters, chosen for appraisal, are the prime objectives of our system; they help us to view each system in the light of our objectives.

2.2.1 PLANES

PLANES is NL Interface for answering queries on a data base of aircraft flisht and maintainance records.

Normal operation of PLANES entails three steps, namely, parsing concept case frame generation, and query generation.

While parsing, a first pass performs spelling correction, substitues roots and inflection markers for inflected words and removes noise phrases; a second pass transforms the query into a set of unordered semantic constituents(like plane-type, date, and time). This is accomplished by using a set of subnets(ATNs) each of which recognise a semantic constituent.

The second step compares the unordered semantic constituents with standard concept case frames. These concept case frames are templates consisting of a sequence of semantic constituents. The matched template or the concept case frame is issued to the next phase. In case some constituents are missing from the standard templates, the required constituents are borrowed from the context set by the previous queries, thus meeting the needs of ellipsis.

The third pass interacts with the DB and generates the query from the concept case frame. A key finding of PLANES is that a set of semantic constituents uniquely decide the formal query. Using this principle, it converts the concept case frame (a template of semantic constituents) into a formal query.

PLANES is not designed to adapt to a new data base; in fact, it uses many assumptions on its domain of discourse, shackling itself heavily (too heavily) to the specific domain of aircraft data base.

The transformation of an unordered set of semantic constituents into a formal query makes an ill organised approach to the prepositional modifiers; this principle fails not only in other domains, but even in its own domain as illustrated below.

The subnets of PLANES recognise the word "hours" as the semantic constituent "flight-hours" and the word "NOR" or "NOR hours" as the semantic constituent "not-operationally-ready-hours". So in the following query,

".... hours of NOR...."

PLANES recognises two semantic constituents while the user intends only one.

Similarly, the language accepted by PLANES constitutes a small subset of NL because it does not capture the syntactic regularity of the language. For a new data base, with a larger subset of NL, it becomes very difficult to use the principles followed in PLANES.

2.2.2 RANDEZVOUS

RANDEZVOUS is a NL interface system for relational data base dealing with suppliers, parts, shipments and projects.

The first phase, the analyzer, converts the user's query into formal query language DEDUCE. This conversion uses a set of rewrite rules which map a specific input pattern into a fragment of DEDUCE query. The conversion is bottom up and non-backtracking. The analyzer resolves ambiguities by a dialogue with the user using the routine called menu-driver.

The second phase, the generator, restates the DEDUCE fragments in precise English and asks user's approval. On failure, the menu driver takes over and the process repeats; on success, the generator passes the DEDUCE fragments to the third phase.

The third phase, the retriever, actually interacts with the data base and retrieves the answer.

RANDEZVOUS runs mainly on the rewrite rules. These rewrite rules are deterministic and non-backtracking resulting in a typically long clarification dialogues for even simple queries.

Designed with the goal of accepting any English query, RANDEZVOUS captures little · syntactic regularity of the query.This results in its failure to answer certain quries correctly. Queries involving more than two linking joins constitute a typical example.

It is not designed to adapt to a new data. base. The rewrite rules are semantic-grammar based. They translate matched input patterns into DEDUCE fragments. But when it is to be ported to a new data base, all these rewrite rules have to be replaced. This involves a complete redesign.

2.2.3 LUNAR

LUNAR is NL interface system for NASA data base of chemical analysis data on LUNAR rock and soil composition.

The first phase, an ATN based parser, produces a syntactic structure of the input query; it is efficient to map many surface forms into thSSe same parse.

The second phase, the semantic interpreter, maps specific syntactic structures into fragments of a formal query. The rules used in this transformation are determined from the verbs, the nouns and the determiners of the input query. Each lexical catesory(like verb, noun) alons with a specific syntactic structure (like noun phrase, verb phrase) in which they can occur, tosether define a set of rules. These rules, derived from the semantics of the data base map the syntactic structures into formal query fragments.

The third phase, the query senerator, retrieves the answer usins formal query senerated by the second phase.

LUNAR uses certain principles that result in a seneral interface. To that effect, it encapsulates syntactic resularity, and modularises syntactic to semantic transformation. But, it suffers from the followins

problems.

The data base used for LUNAR is a flat file. Consequently, it can not be ported to a data base which contains entity relations and linking relations. Such data bases involve identifying the particular relation of a recognised field, performing operations like selection, projection, and join. These operations are not accounted for, at all, in LUNAR.

In LUNAR, no single element can appear in more than two domains(like phases of analysis, chemical constituents, units of measure, etc.). So, it does not have the problems of finding an implied relationship between one noun and another that qualifies it. If there are two nouns and their domains are found, then the relation between them is fixed and unique. Such a restriction severs the set of data bases which can adopt the principles of LUNAR. This is because, in many data bases, a single field can occur in more than one relation; even if the relations of a pair of fields are identified, the way they are related to each other, through other fields is not unique. Such intelligent access of relations is not considered, at all, in LUNAR. So, it can not be ported to any practical data base.

LUNAR keeps every proper noun in its lexicon. Since the different types of entries is small, it could use such a scheme. If we contrast it with the data base of personnel records having names of persons, street and lane addresses, we realise how significantly the size of the lexicon increases. This is another restriction in applying the approach followed in LUNAR.

2.2.4 LADDER

LADDER is desigred to answer queries on US Naval command and control data base.

The following is a brief description of how LADDER converts its English query into a formal query.

The first phase, INLAND, takes in restricted English queries and produces skeleton queries. It identifies the fields of the data base that are specified in the query. It does not mention the various relations to be accessed, joins to be made and other such operations to be performed.

The second component, IDA, breaks the skeleton queries into sub queries indicating the relations to be accessed, and the linking to be made. These subqueries are passed to

the next phase.

The third phase, FAM, decides what files to access and finally retrieves the answers from the data base.

LADDER uses LIFER as the parser. To appraise LADDER's linguistic power, we need to know the principles of LIFER.

LIFER is a semantic-grammar based parser generator. It is independent of any particular data base. It consists of a set of interactive functions for specifying the language fragment to be used for a specific application. These interactive functions define productions of the following format.

<meta-symbol> =====> <pattern>, <expression>.

where <meta-symbol> is a meta-symbol of the language, (like semantic constituents of the data base for which it is used), <pattern> is a pattern of input language symbols and meta-symbols and <expression> is a LISP expression whose value, when computed, gives the value to be assigned to <meta-symbol>.

The input is parsed left-to-right, top-down. Whenever a meta symbol is recognised, corresponding parse tree is constructed and the <expression> associated with the <meta-symbol> is evaluated and put along with the <meta-symbol> in the parse tree. When the tree is finally completed, the value of the meta-symbol representing the root of the tree is the formal query. LIFER's meta symbols are not syntactic structures(like noun phrase, verb phrases). Instead, they are semantic constituents of the specific data base(like shipname, ship-property in the case of LADDER). LIFER encourages the interface designer to specify a language which encodes the syntax of the input language into the db-constituents.

We shall now see the problems involved in the portability of LADDER. These problems are inherent in any semantic-grammar based approach. Hence they are carried to LADDER because it uses LIFER - a semantic parser.

LIFER does not carry the description of language from one application domain to anotheras it does not capture the syntax at all. So whenever a data base is changed, a new set of semantic productions are to be designed. This is no mean a task and no less than a total redesign.

The design of the semantic rules for a new data base forces the designer to map input language patterns into formal query fragments. This encoding of input to output language is done in a single step by the productions. This requires high level of expertise in understanding the content of the data base and the structure of the potential English patterns, hindering an easy switch to a new data base. As the input language is expanded, these productions become difficult to conceive and complex to implement and their adhoc character eventually renders them unusable.

Since little attention is paid to syntax, the language accepted is a small subset of NL and any extension, as discussed above, is difficult.

From this survey we can draw the following conclusions.

Each of the systems discussed above are designed for a specific data base; their tolerance to a change in data base is small.

PLANES and RANDEZVOUS are just glued to their respective data bases while LUNAR and LADDER use some principles that are general and independent of any data base in particular. However, LUNAR's assumptions on its

simple(too simple) data base do not warrant portability to any practical data base, while LADDER's semantic-grammar based parser renders it a poor and inextensible language. More over a change in the data base, in LADDER, involves a complete redesign of the semantic productions. These problems render both the systems unadaptable to a new data base.

Nevertheless, LUNAR's language power is high. This is mainly because it captures the syntactic regularity of the language thus enabling it to accept a large subset of NL. Similarly, LADDER's data base access is powerful. It interprets missing joins by using a schema derived from the data base. Also it makes intelligent access to data base.

Our aim, as formulated in chapter 1, is to design a system that has both intelligent data access and habitable NL power. Added to this it is expected to be easily portable from one data base to another. The extent to which we met these aims is discussed in chapter 7.

2.3 ATNS AS PARSER

Ausmented Trasition Network(ATN) is a formalism to parse NL. Its main appeal is its simplicity, which made it one of the most common methods of parsing in many systems of NL interface(like LUNAR, PLANES). We use ATNs to recognise the field descriptions and to parse the query.

We first describe transition networks and then see how ausmented transition networks are built from them. Finally we give the way our formalism deviates from the standard formalism.

2.3.1 TRANSITION NETWORKS

A transition network is a finite state machine. The arcs of a transition network indicate a linguistic category to which the current input word should belong.

An arc is traversed if the current word of the input sentence belongs to the category indicated on the arc. When one arc is traversed, exactly one word is "consumed". When the final arc "done" is encountered, the sentence should have been completely consumed. Else transition network returns a failure indicating that the sentence can not be

accepted.

The Fig 2.1 shows a simple transition network which accepts simple sentences. Consider "the dog bites the cat". The arcs that match the input are 1,2,4,5,6.

2.3.2 AUGMENTED TRANSITION NETWORKS

A Transition Network augmented with two facets evolves into Augmented Transition Network(ATN). The first one is the introduction of the concept of a "Register", the second one is extended meaning of "arc".

Each ATN is associated with a set of registers. A register is an ASSOC list of the following format.

(<label> <value>)

where <label> is the name of the register and <value> is the value set to the register.

An arc in a Transition Network indicates only lexical categories(like verbs, nouns) whereas the arcs of a ATN can be a syntactic categories(like noun phrases, verb phrases) as well. The arc labelled with a syntactic category is

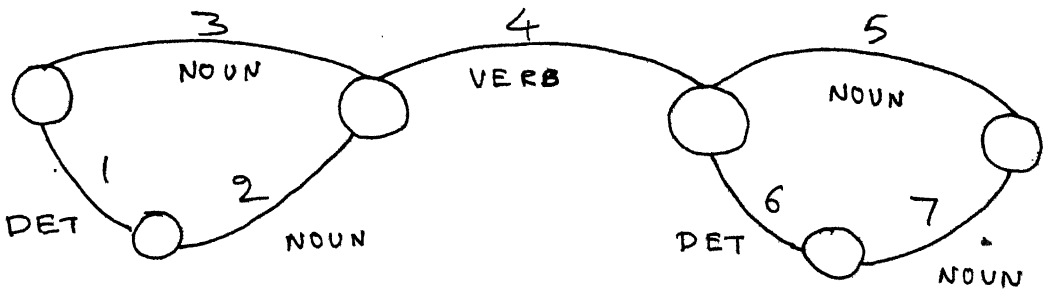


Fig 2.1. A TRANSITION NETWORK

<LABEL>
<NAME> <VALUE>
<NAME> <VALUE>
<NAME> <VALUE>

Fig 2.2. A REGISTER FRAME

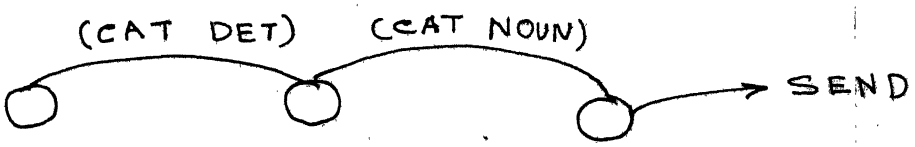


Fig 2.3. An ATN for NOUN PHRASE.

traversed if the next few words match the ATN defining syntactic category. Another change in the "arc" is introduction of explicit conditions. In the Transition Network, the only condition on an arc is that the current word should belong to the category indicated by an arc. This is called implicit condition. The implicit condition in the arcs of an ATN is that the next word belongs to the lexical category(if the arc indicates a lexical category) or a syntactic category(if the arc indicates a syntactic category). Besides this implicit condition, an arc of an ATN may have explicit conditions which must be satisfied to traverse the arc. These explicit conditions are called simply "conditions". A third extension to the arc is "actions". In an ATN, when an arc is taken, the corresponding parse structure, if any, has to be built. To do so, each arc may have some "actions" which produce the required side effects to construct the parse structure. Thus, including these three changes, a typical arc of an ATN looks as below.

"<condition> <arc-type> <actions> <next-node>"

where <arc-type> indicates whether the arc recognises a lexical category or a syntactic category (or some other type

as discussed later), <conditions> is the explicit conditions to be satisfied for a successful traversal of the arc, <actions> is the actions to be carried on successful traversal, and <next-node> indicates the the node where the control transfers after the arc is traversed.

We now present a complete description of ATN formalism.

1. It consists of a set of ATNs. One of them is considered distinguished and is normally given the name "S".

2. An ATN is a 4-tuple < label, states, arcs, registers>
comprising of

the name of the ATN,

the set of states,

the set of arcs,

and the set of registers.

The registers are of two types, feature registers, and role registers.

The feature registers are used to store the features of the words recognised. For example, if an arc recognises a noun, then the actions of the arc may set the feature register "number" to "singular" or "plural" according as the word is singular or plural. Similarly, another feature register could be named "person" which is set to the person(1st, 2nd, 3rd) of the noun. On recognising the word "kicks" the following feature register set may be set to the appropriate values.

(word kicks)

(tense present)

(voice active)

(number plural)

A role register contains, as its value, a register frame. A register frame consists of

1. label indicating the name of the frame,

2. a set of registers denoted by their name, value pairs.

A register frame is shown in Fig 2.2. The definition of

register frame is recursive because a role register which is a constituent of a register frame contains, as its value, another register frame.

For example, consider Fig 2.3 showing an ATN to recognise a noun phrase. The first arc checks for "det" and second arc checks for a "noun". When an arc is successful, the actions associated with it construct a register frame consisting of the feature registers of that word. When the ATN is successfully traversed, the actions on the last arc construct a register frame which contains each of the constituents recognised.

Let "the boy" be the input to the ATN of Fig 2.3. When "the" is recognised as a "det", the relevent feature registers are constructed as shown in Fig 2.4.

Similarly, the role register "noun" is also constructed with relavant feature registers as shown in Fig 2.5. At the end of the ATN, the role register consisting of the constituents recognised are as shown in the Fig 2.6.

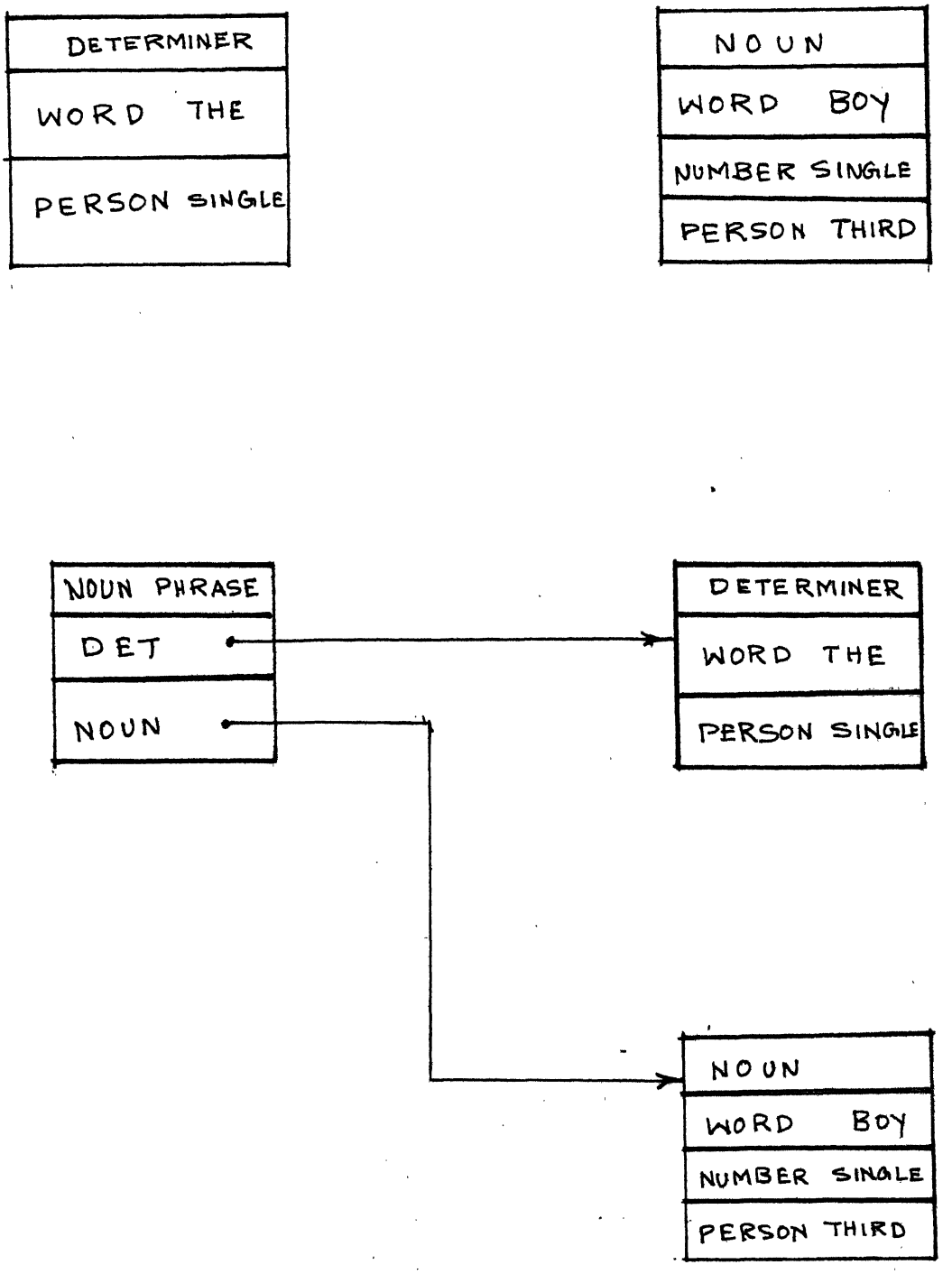


Fig 2.6. REGISTER FRAME FOR NOUN PHRASE

THE ARCS

The followins arcs are common in standard ATN formalism.

1. The word arc:- This is taken when the current word is the same as the one specified in this arc.

2. The CAT arc:- This arc is taken if the current word belonss to the lexical catesory specified by this arc.

3. JUMP arc:- This arc is taken unconditionally.

4. SEEK arc:- This arc specifies a syntactic catesory which must be matched by the current word and next few words. So, this arc is traversed if the ATN, correspondins to the syntactic catesory specified in the arc, is successful.

5. SEND arc:- This arc returns, with success, the structure parsed in that ATN. The control soes to the ATN which called it.

We shall illustrate with an example. Fis 2.7 shows an ATN srammar with two ATNs for simple sentences.

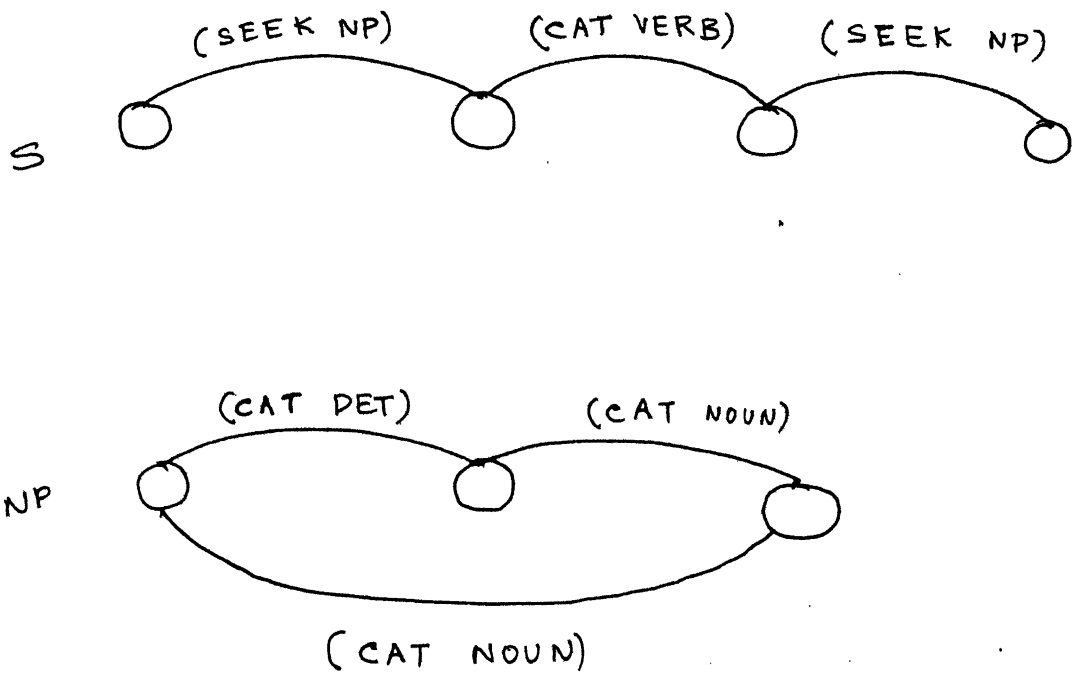


Fig 2.7. An ATN GRAMMAR.

The conditions on all the arcs are kept "true" except on the arc 2. The condition on this arc checks whether the value of feature register "number" of the role register "NP" of the previous arc agrees with the value of the feature register "number" of the present arc. This is to allow sentences of proper concord only.

Consider the sentence "boy kicks the ball".

The first arc in S is a SEEK to NP. So, the control is transfered to the ATN NP. It parses "boy" as a noun-phrase returning the structure shown in Fig 2.8.

The next arc in S is a category arc which consumes "kicks" and sets the feature registers as shown in Fig 2.9. The condition on this arc is satisfied. So, this arc is traversed and we reach third arc. This is a SEEK arc. So, control goes to the ATN NP and it parses "the ball" as a noun phrase and returns the role register, shown in Fig 2.10, to the ATN S. By then, the sentence is completely consumed, and the SEND arc of S returns the role register shown in Fig 2.11

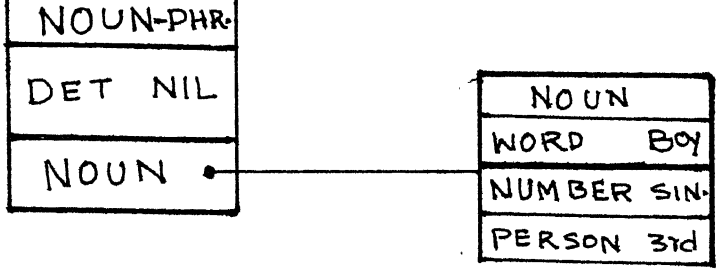


Fig 2.8. Register Frame for "boy" as noun phrase

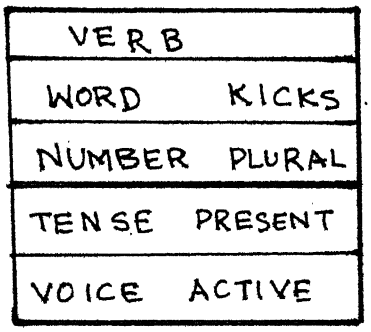


Fig 2.9. Register frame for "kicks" as verb

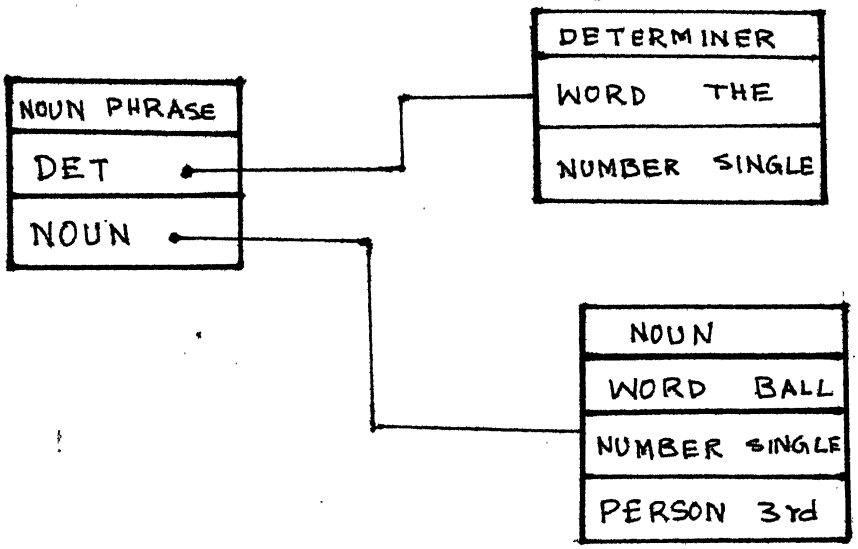


Fig 2.10. Register Frame for "ball" as noun phrase

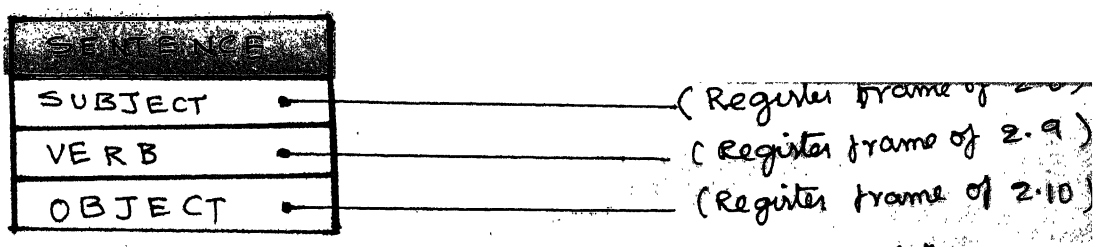


Fig 2.11. Role Register for "boy kicks the ball".

2.3.3 OUR FORMALISM

Our formalism differs slightly from the standard formalism to meet our needs more efficiently. Our system does not "understand" the English query in order to convert it into formal query. So, a deep structured parse is not needed. Detailed features like "number", "person", "gender" are also not needed. This allows us to avoid knowing every feature of each word. The only features we are intersted in are the lexical category of the word and the data base specific information, if any. This renders the feature register concept un necessary for our formalism. The structure to be built on recognising any word is

(<lexical category> <data base specific information> <word>)

This structure is independent of <lexical category>, whereas in the standard formalism the structure depends on the <lexical category> For example in the case of "verb" a feature register "tense" is there whereas in the case of "noun" a feature register "person" is there. In our formalism, there is no such change in structure. To construct such fixed format, we do not need special "feature

resister" concept.

Similarly, the structure to be built at the end of any ATN is also standardised as follows.

(<atn-name> (LIST of syntactic structures led to success))

Such a structure can be built while traversing the ATN and we do not need, for such simple structure, a detailed role resister concept.

Another place where resisters are used is in the <conditions> on the arcs of a ATN in the standard formalism. These conditions check the resisters of the previous arcs. This is met in our formalism as follows. The conditions on arc in an ATN can access the structures recognised by previous arcs of the ATN. So, they can access any struture of the previous arc, and check any feature(only two features are there in our formalism as explained earlier). In fact, the conditions on the arc can access any structure of the over all parse tree, althoush such inter-ATN communication is seldom needed in our formalism. This is because, we are buildins a shallow structure unlike the standard ATN formalism which builds deep structure.

The actions associated with any arc in the standard ATN formalism typically build the parse tree. In our formalism, the parse tree structure is defined independent of the category of the word and independent of the ATN, unlike the standard formalism. For example, if a verb is recognised in standard formalism, there is one type of register frame constructed while another type of register register is constructed for a noun. In our formalism it is not the case. The parse structure is independent of the type of the word. Hence, we pushed "actions" into the interpreter thus avoiding explicit mention of these actions. When an arc(that needs to build a parse structure) is taken, then the actions are automatically carried out by the interpreter. Thus, when a category arc is taken the following structure is automatically constructed.

(<cat-name> <data base specific info> <word>)

This structure is the same whatever <cat name> is. Thus we avoid explicit actions on the arcs.

So, in our formalism, the structure of an arc is as follows.

(< pre-condition> <next-node>)

where <pre-condition> in the simplest case can be a arc-type. The pre-condition is EVALuated and on success(non NIL), <next-node> is reached. If there are conditions on the arc, then the <pre-condition> looks as below.

```
(and [condition 1]
     [condition 2]
     [condition 3]
     ............
     <arc-type>       )
```

This is valid because, in our formalism, the <arc-types> are defined as LISP macros or functions, thus allowing the user to define his own arcs. Such a unified approach to <arc-type> and <conditions> avoids the unnecessary case checking in the interpreter(ATN interpreter) and also facilitates an easy definition of arcs by the designer. In the standard ATN formalism, if a new arc is to be defined, it has to be incorporated in the interpreter as another case statement. This increases the potential for errors in the interpretor and also increases case checking. In our formalism, what all to be done is to define a new function(or a macro) with the name of the arc. In case the arc has to build a parse tree, then these side

effects are incorporated in this function definition itself. These side effects are carried out automatically when the arc(a funnction or a macro) is EVALuated.

We shall describe the arcs of our formalism.

ARCS

All of them have the followins format by default.

(<arc-type> <arsument>)

The arsument is not evaluated unless otherwise stated. <arsument> is only one atom unless specifically mentioned.

1. PARSE arc :- The <arsument> sives the name of the syntactic catesory into which next few words have to be recosnised. This is achieved by a call to the ATN which recosnises the syntactic catesory. It is same as SEEK arc in the standard formalism.

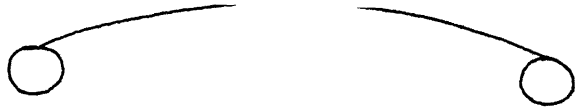
2. DONE arc :- This has the structure "(done)". This arc is an indication of successful completion of the ATN. So, it completes the actions to be done on successful completion of an ATN. The structure thus formed is returned to the ATN which called it. It is same as SEND arc in the standard ATN formalism.

3. T arc :- This is a JUMP arc which is taken unconditionally. It does not consume any input.

4. WORD arc :- This is taken if the next word belongs to the category which is indicated by <argument> of the arc. The "category" is lexical category or part of speech. This is same as CAT arc.

5. CHECK-FOR arc :- This has an indefinite number of <argument>s. It checks for the condition that the next input word is the same as any of the words passed as <argument>s to it. If the condition returns success, it is taken. It does not evaluate the <argument>s. Fig 2.12 shows it.

6. CHECK-LIST arc :- This evaluates its argument which should give a list. Then it checks if the next word is a member of this list. If so, the arc is taken else, it fails to take the arc. It is shown in the Fig 2.13 .



INPUT: in the.....
↑
SUCCEEDS

Fig 2.12. Illustrating "check-for" arc

COURSES = (DATA STRUCTURES COMPILER)

INPUT: Data structures - - - -.
↑

SUCCEEDS

Fig 2.13. ILLUSTRATING "CHECK-LIST ARC"

INPUT: in the hall - - - - -

SUCCEEDS

Fig 2.14. Illustrating "SEEK" arc

7. SEEK arc :- This takes two arguments; the first one is a number (an integer) and the second is a word(not evaluated). The arc is a look ahead arc. It succeeds if the second argument, namely the word, is in the first n number of words of the input sentence. Here n is the integer specified with the first argument. Fig 2.14 shows it.

8. SEEK-LIST arc :-

It is also a look ahead function. It has the structure

"(seek-list list number)"

This arc evaluates the first argument to get a list. The arc succeeds if any word of the list so obtained is there in the first n number of words of the input sentence. Here n is determined from the number given in the arc. Fig 2.15 illustrates this.

CHAPTER 3

SYSTEM OVERVIEW

The prime objectives of our system, as formulated in chapter 1, are acceptability of habitable NL and DB-independence. In this chapter, we see how an attempt to honour our objectives forces certain design considerations which inturn structure the system. The various modules so developed are identified and discussed in brief. A detail discussion of each module follows in later chapters.

The last section discusses the implications of accepting a NL query and forms the limitations of the NL accepted by our system.

3.1 DIFFERENT APPROACHES TO NLI

Almost all the NLI systems can be broadly classified into three catesories accordins to the type of the srammar they accept.[Rich84] They are

(i) Languase throush windows,

(ii) Semantic Grammars, and

(iii) Syntactic Grammars.

We have chosen the third; we digress a little here to justify our choice by discussing the relative merits of each of the above.

3.1.1 LANGUAGE THROUGH WINDOWS

It is similar to QBE + NL flavour. The system opens some windows to the user on the screen. When the user specifies the values of fields which he knows, and indicates the fields which he needs, then the systems gives the output. An example is NLX, designed by Texas Instruments.

They are relatively efficient but the options available to the user are very rigorously constrained. Secondly, the user should know the organisation of the DB. These go against our objectives of a habitable NL and accessibility to a novice user.

3.1.2 SEMANTIC GRAMMAR APPROACH

This is used by some practical systems like LADDER[2] and PLANES[3]. We proceed with an example to discuss what it means.

Consider LADDER system for example; it is desisned for US Navy. A small subset of the semantic srammar productions used by it are siven in Fis 3.1.

```
S ----------------->    QUERY SHIP-PROPERTY of SHIP

QUERY ------------->     what is/ tell me

SHIP-PROPERTY --->        the SHIP-PROP/ SHIP-PROP

SHIP-PROP -------->     speed/lensth/tupe

SHIP ------------->       the SHIP-NAME/ SHIP-NAME

SHIP-NAME -------->        Kennedy/ KItty hawk/...
```

Fis 3.1 A subset of LADDER's srammar.

The following questions

"what is length of Kennedy"

"what is speed of Kitty hawk" etc.

can be very easily parsed by the rules of the grammar of Fig 3.1. Each grammar rule or production has a set of actions transformation rules(trans-rules)' which are applied whenever the grammar rule is applied. These trans-rules produce the Formal-query. The Fig 3.2 shows how the formal query is produced from the sentence by using both the parsing rules and the trans-rules. The nodes of the tree in the Fig 3.2 indicate the structures into which the query is parsed. Whenever a syntactic structure is recognised, a variable "V" is set to a data structure obtained from the syntactic structure using the transformation rules. In the figure, when "kennedy" is recognised as a SHIP-NAME, then V is set equal to '(NAME EQ JOHN F KENNEDY). This structure is derived from

(i). the syntactic structure encircled in the Fig 3.2

(ii). the transformation rules put in words like "NAME","EQ" and thus complete the name of the ship. Similarly, other structures are also recognised.

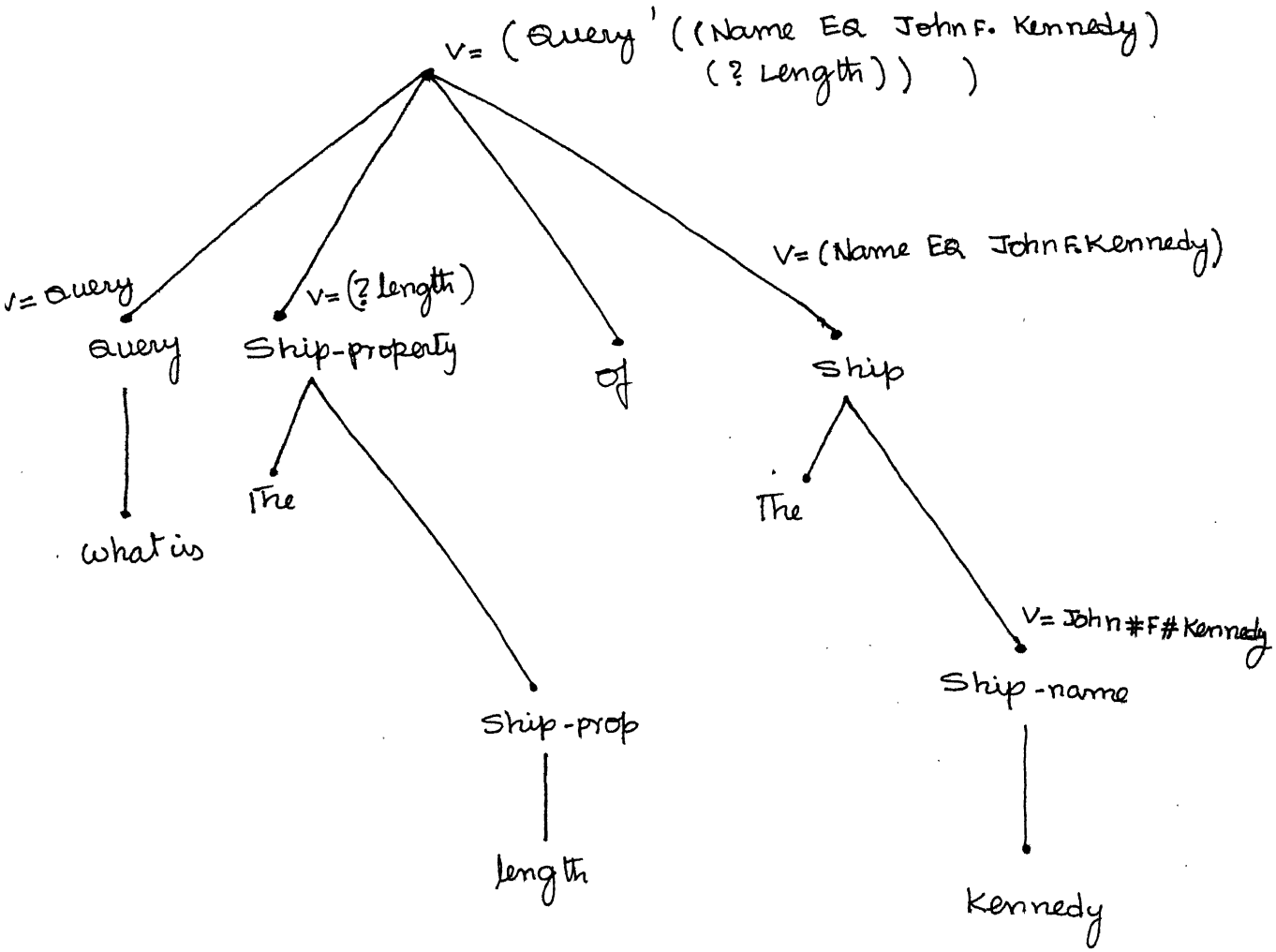


Fig. 3.2. Parse tree for "what is the length of Kennedy".

```
(S    (Int-Pro    who)
      (VP (Verb   taught))
      (proper-noun (cname systemsprogramming))
)
```

Fig 3.4. Parse for the query:
"who taught systems programming".

The main differences between this and an ordinary English grammar is that the categories which this uses are not formal English syntactic categories but, those designed specially for the the interface. They are based on the semantics of the DB.

There are many problems with this approach.

* It is useful only when the language to be accepted is a very small subset of NL. As it does not capture the syntactic regularity of the language, it is more difficult to design the grammar a large subset of NL. Eventually their ad hoc character makes them unusable.

* It is designed keeping in view both the input language and the target program (or target language). The grammar rules and the trans-rules are designed in such a way that they directly map the structures of one into the other. However, as the specifications of the input language becomes larger, it becomes very difficult to conceive the process in a single step. So, the design of the grammar becomes very difficult.

* Whenever the DB is changed, a new semantic grammar and along with it the trans-rules have to be designed. This means an entire redesign of the system.

* It does not capture the language structure. They are based on matching certain predefined patterns.

All these go against the major objectives of our system.

3.1.3 THE SYNTACTIC GRAMMAR APPROACH

In this approach there are two steps. The first step parses a given input NL query according to certain rules of grammar. The second step takes the parsed output and applies heuristics using DB structure and produces the formal query.

The grammar used in the first step is made general enough to handle a large class of input queries. While it is designed with the knowledge that it will be used for parsing of DB queries, and in that respect might differ from conventional grammars of NL, it is independent of any particular DB.

The second step which takes the parse and produces the formal query is called Formal Query generation. The heuristics used in this phase utilise the syntactic structure of the parse, the meanins of certain DB specific words and the relational structure of the DB to produce the formal query.

Let us see with an example.

Ex1. consider the grammar shown in the Fis 3.3

S ----------> INT-PRO VP NP

INT-PRO----------> who/what/which

VP ----------> VERB/ TO-BE GERUND

NP ----------> DET NOUN/ NOUN/ PROP-NOUN

VERB ----------> teach/ offer / sive/ take/....

TO-BE ----------> is / was / are/ were/ ..

NOUN ----------> teacher/ teachers/ ....

GERUND ----------> teachins/ offerins/...

PROP-NOUN ----------> CNAME/ SNAME/ TNAME/....

Fig 3.3 A syntactic grammar

The query

"who taught systems programming"

has the parse shown in Fig 3.4.

We define the following rules(for the grammar of Fig 3.3) to generate the formal query from the parse structure.

(i). Using INT-PRO and VP identify the unknown. The verb in the VP or Gerund in the VP conveys information regarding what the needed field is.

(ii). Using the NP decide the known or specified field.

Since the word "taught" indicates that we want "TNAME" and NP indicates that we are given CNAME, a list is generated as shown below. The first element indicates that TNAME is needed and second indicates that CNAME is given.

((? TNAME)

(= CNAME (systems programming)))

Followins are some more examples which satisfy the grammar and can be processed by the above rules.

"who is teachins kumar"

"who are takins datastructures"

"who takes computerorsanisations"

This approach is followed because of the followins reasons.

* A larse subset of NL can be accommodated by this.

* The syntactic resularity in NL is captured unlike other approaches.

*The srammar is desisned with the knowledse that it will be used for parsins DB queries. However, it remains independent of any particular DB.

* Trans-rules depend only on the grammar and hence are independent of any particular DB.

We now see that the system has at least two main parts.

The first part produces the parse of the sentence. The parsing technique followed is ATN formalism which is discussed in chapter 2. As with any other parser, a lexicon is kept along with but separate from parser. It contains the words usable in the queries along with their syntactic categories. The considerations in designing the grammar, the logical organisation of the lexicon are discussed in detail in chapter 4.

The second part applies heuristics to the parse structure and produces the formal query. This pass is called Formal Query generation and is discussed in the next section. A block diagram of the system consisting of these two parts is shown in Fig 3.5.

3.2 FORMAL QUERY GENERATION

To understand the basic steps in this process, consider the following query.

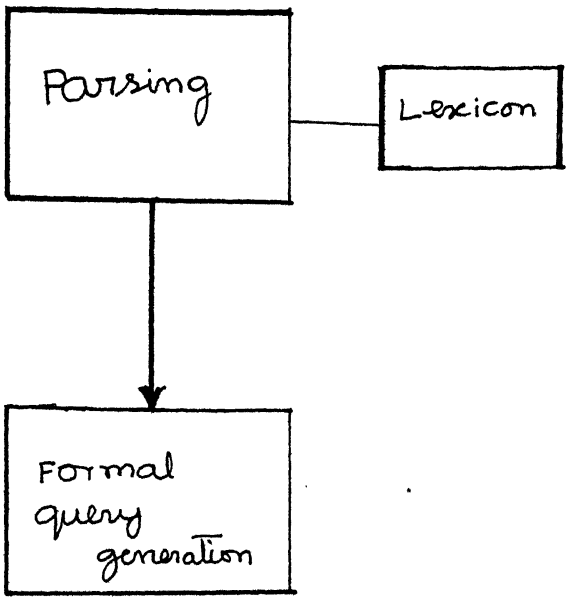


Fig. 3. 5. Block diagram of the system showing the primary parts.

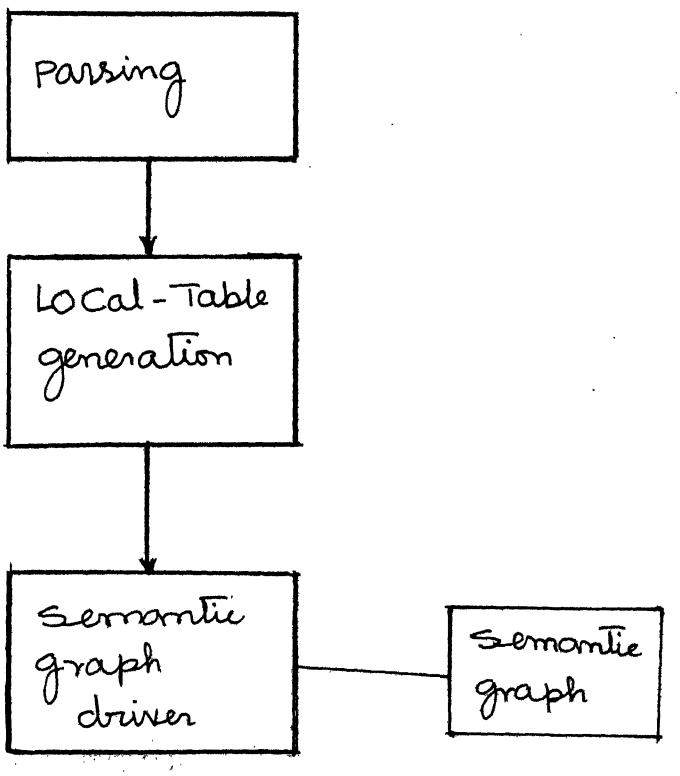


Fig3.7. Block diagram showing the

"who taught systemspros"

This conforms to the grammar described in Fig 3.3. The parse is obtained as below:

```
(       ( INT-PRO who)

        (VP (VERB taught))

        (NP (PROP-NOUN ( CNAME syspros)))          )
```

The first step in formal query generation is identifying the fields, deciding on what fields are needed and what fields are given etc. To get such information, many heuristics are used in this phase. These heuristics operate on the structure of the parse and also on the DB-specific information of certain words used in the query.

In the present example we proceed as below.

(1). From the interrogative pronoun and VP decide what is the needed field.

(2). From the NP decide what is the given field.

So, in the above example, the verb "taught" and the INT-PRO "who" indicate that the needed field is "TNO". This is obtained from the DB-specific information kept with the word "taught" in the lexicon and from the fact that "who" is personal INTPRO.

So, the needed field is TNO and the given field is CNAME. It is such a kind of analysis that is done in the phase.

The information regarding what fields are needed, what fields are given is stored in a data structure called skeleton structure. The skeleton structure of our example is shown below.

```
(       ( ?   TNO)
                    ( = CNAME (SYSTEMS PROGRAMMI
```

where the first structure indicates that TNO is needed and second structure indicates that CNAME is given.

In the second step of this pass, the skeleton query is converted into the formal query. This involves identifying the relations containing the given fields, and issuing out expressions indicating the necessary select, project, and

Join operations to be made.

Let us assume the DB structure shown in Fig 3.6.

The formal query generated is as shown below. We save the formal query in English like statements. The actual syntax of the formal queries is discussed chapter 6.

1. Instantiate CNAME to "(systems program)".

2. Select the tuples of the relation COURSE which have their CNAME field as instantiated already.

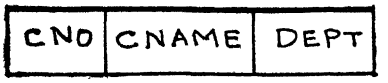
3. Project their CNO field.

4. Join these into the relation OFFER on CNO field.

5. Project the TNO field of these tuples.

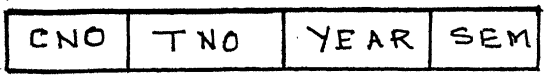
Thus, we see that formal query generation has two distinct parts. The first part takes the parse of the query and identifies various fields, decides what fields are needed and what fields are given, and finally constructs the skeleton structure. We call these skeleton structures as LOCAL-TABLES. The generation of these LOACAL-TABLES

TEACHER

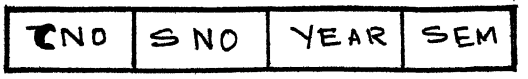


COURSE

STUDENT



OFFERING



CREDIT

Fig 3.6. AN EXAMPLE DATABASE

involves heuristics which operate on the syntactic structure of the parse and the DB-specific information of the words. This process is discussed in detail in chapter 5.

The second step takes skeleton structures and produces the formal query which indicates what relations are to be chosen, what fields are to be projected, selected, and joined. This step utilises the information of the relational structure, key fields, and linking information. All this required information is mapped into a graph called semantic graph. This graph is used to generate the formal query and so this process is called semantic graph driver which is discussed in detail in chapter 6. A block structure of the system showing

"Formal query generation" into two parts, namely local table generation and semantic graph is shown in Fig 3.7.

3.2.1 ORGANISATION Of LEXICON

We have seen above that the heuristics used in the local-table generation use the syntactic as well as DB-specific information of the words of the query.

For example, consider the query

"who taught cname = (data structures)"

The needed field of the above query is TNO. To get such information we use the fact that "taught" refers to TNO. The best place to keep all such DB-specific information is lexicon.

So, we introduced another slot for each word besides the usual slot <lex-info>. This extra slot contains the information regarding the DB-specific information. During the parse tree construction, the In the parse tree, along with each word is kept not only its lexical category but also the DB-specific information. local-table generator an easy access of information.

The organisation of the lexicon is discussed in detail in chapter 4.

3.3 RECOGNITION OF FIELD DESCRIPTIONS

There are many alternative ways to describe fields in a NL query. To facilitate the parser to recognise such different descriptions of a field into one structure, we

replace the field descriptions by a standard format called canonical form.

Let us illustrate it with examples.

who is teaching data structures

who is teaching course of data structures

who is teaching data structure course

All the above queries have the same structure except for the difference in the description of particular field namely course-name field.These must have the same parse. We identify such structures and replace them by a cannonical form. This pass is called Normalization and it is done before parsing. The cannocical form in the present case is

"cname = (data structures)"

If this cannonical form replaces the description of the field in the query, then the above queries reduce to

"who teaches cname = (data structures) "

producing the same parse structure. The normalizer process needs information about various ways of describing a particular field. So, we collect together all the possible descriptions of each DB-field into a group collectively called Filter Networks. A driver routine Normalizer is used to compare the query with the descriptions given by the filter networks and if any field description is identified, it replaces the field description with the cannonical form. The description of Filter Networks and the Normalizer are discussed in detail in chapter 4. As we have already seen normalization should be carried out before parsing. The system structure including the normalization is as shown in Fig 3.8.

3.4 A COMPLETE EXAMPLE

We conclude this section by giving an example which shows how various blocks in the Fig 3.8 effect the proces of producing the formal query from the English query.

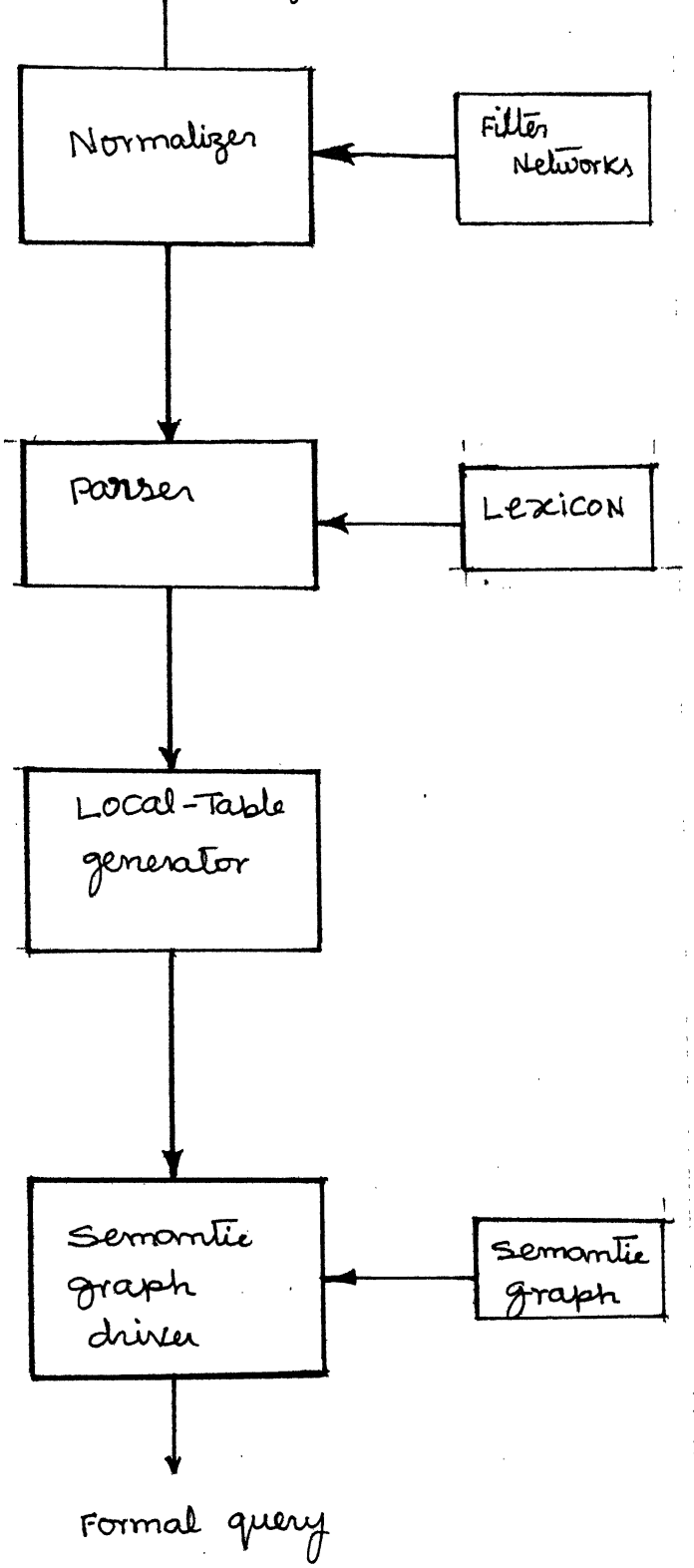


Fig. 3.8. Complete block diagram of our system.

"who taught a course in the department of computer science "

I. NORMALIZATION:

The underlined string of words is recognised to be the field description of the field "dept". By replacing it with the cannonical form we get the following.

"who taught a course in the DEPT = (computer science)"

II. PARSING:

Besides recognising the linguistic structures involved in the above query, the parser also keeps some DB-specific information of certain words. The parse looks as below

```
(MAIN-CLAUSE (INT-PRO (PERSON) WHO)
             ( VERB-PHRASE  ((SUBJECT TNO) (OBJECT SNO))
                                              TAUGHT)

             ( NP (DET A ) (NOUN (CNO) COURSE)          )
             (QUALIFIER (NO-INFO)   TO            )
             ( NP  (NOUN (DEPT) DEPT )    )
             (COMPARE (EQ  (NO-INFO)    =  )
             (PROP-NOUN (DB-FIELD) (computer science)))
```

The grammar which produces this parse is not discussed here but it is enough if we assume that the required grammar is there in the parser. A detailed discussion of the grammar is in chapter 4

III. LOCAL-TABLE GENERATION

This pass has all the rules which indicate how we use the linguistic and the DB-specific info to set the fields. The rules depend upon the grammar.

This produces the output as shown below.

```
( LOCAL-TABLE (NEED TNO)

        (VARS

                ( TNO

                        ( CNO

                                (= DEPT (computer
                                   science))
                        )
                )
        )
```

)

This table sives the information that (i) we need "tno" (ii) the instantiated field is "dept" (iii) that "dept" should be used to decide "cno" which inturn should be used to determine the "tno"; this is the heirarchy amons the various fields.

IV. SEMANTIC GRAPH DRIVER

This sives the final form of the query ; it supplements the intermediate fields, if necessry.

Assume the relational structure, the keys and the linkins info as shown in Fis 3.6. The formal query, shown here, consists of English-like statements. The actual formal query syntax is discussed in chapter 6.

Formal query

1. Instantiate "dept" to "(computer science)".

2. Select from the relation COURSE those tuples whose "dept" field is already instantiated.

3. Project the "cno" field of the above tuples.

4. Join these into the relation "OFFER" on "cno".

5. Project the "tno" fields of these tuples.

3.5 LIMITATIONS OF THE NL ACCEPTED

Normal English queries are accepted subject to the following conditions.

* the language should include those English queries with minor errors in concord and tense.

* the query should be structured so that fields can be retrieved unambiguously.

The restrictions stipulated by the above criteria on the language are given below.

3.5.1 Error In Referencing

Inspite of repeated suggestions of orthodox grammarians, the usage of lose structure and improper dereferencing continues to be a part of our life.

Poor structure:: who taught a course in 83 2nd sem which offered by the dept of cs.

Better structure:: Who taught in 83 2nd sem a course which is offered by the dept of cs.

or

Who taught a course which is offered by the dept of cs and which is taught in 84 2nd sem.

The system accepts the second type of construtions but not the former. The rule to be followed to construct such a structure is

" The relative pronoun should be as close to the referent as possible and if there are more than one clause referencing the same referent, they must be connected by conjunctions".

3.5.2 QUALIFYING A PROPER NOUN::

A proper-noun is considered to be a value of a field (of the db) by the system and so, qualifying it by a clause is considered erroneous.

Consider the example

Not accepted :(referencing a proper noun)

"who taught kumar who is a student of the dept of computer sciences"

Acceptable way::

"who taught the student who is having teh name as kumar and who is in in the dept of computer science"

To avoid the structures of the former type, following rule can be used.

"whenever there is a clause specifying more about a proper noun (or precisely whose referent is a proper noun), then replace the proper noun by the appropriate noun and introduce a new clause instantiating that noun to the proper noun and connect the new clause to the specifying clause by a conjunction".

3.5.3 NOISE WORDS

No noise words are allowed in the system language. The exmaples are "tell me", "please", etc.

All the queries accepted by our system are "wh" type.

Here, the "value also specified " in the yes/no query is cno = cs415. So replace it by "a course" and introduce a new clause instantiating "a course" to "cs415". Finally replace yes/no question by wh question. So, it becomes as follows.

"what is the course offered by the dept of cs and which is cs415"

3.5.5 ACCEPTABLE ERRORS

The system accepts some small errors in concord and articles. Examples are as below.

"who are the teacher of cs412"

(error in concord)

"who are a teachers of cs512"

(error in article)

These aspects provide the bounderies to the language accepted by our system. A detailed discussion of the grammar accepted by the system is given in chapter 4.

CHAPTER 4

NORMALIZATION AND PARSING

4.1 INTRODUCTION

Although normalization and parsing, as introduced in chapter 3, are two different passes, both are combined into a single chapter because of the similarities in the formalisms used for both of them. Each is discussed in detail in the sections that follow.

4.2 NORMALIZATION

The design considerations that led to the decision to normalize a query before parsing it are discussed in detail in section 3.3 A brief recapitulation is given here.

consider the queries

"who taught the course of data structures in the dept of cs"

"who taught data structures in the dept of computer science"

Both the above queries have the same structure except a change in the descriptions of the fields "cname" and "dept". In order to facilitate the parser to recognise such variations in the field descriptions and to map them to the same structure, we decide to replace the strings of descriptions by a unique representation scheme as follows

"(fname relation-operator value)"

where "fname" is the field name relational-operator is =,<,> etc, and value is the value that is provided by the description.

In the above example, the field description of "course" may be represented by using schema as follows.

"cname = (data structures)"

Similarly, the string describing the field "dept" can be replaced by "dept" = (computer science)"

Replacing the field description with the cannonical form, we get the following query.

"who taught the cname = (data structures) in the dept = (computer science)"

Such a unified representation scheme which replaces the English description is what we call "cannonical form". In the example above, "cname = (data structures)" is the cannonical representation. The query so developed after replacing all the descriptions is called "normalised query" and the process is called "normalization".

The objectives of the normalization process are

* to accept the field descripions specified by grammar. This allows arbitrary descriptions to be defined by the user.

* to generate cannonical representation of each field description.

There are two main parts in the normalization process.

(i). A data driven program called normalizer

(ii). Data structures containing the field descriptions and the actions which are taken on successful field recognition; these actions generate the cannonical representation. These datastructures are called filters and the actions and the filters together are called filter networks.

The data driven normalizer takes the input query, scans word by word and compares with the field descriptions. When a field description is found, it performs the actions corresponding to the recognised field. These actions replace the description with the cannonical form. After the query is completely scanned, the normalised query is returned.

4.2.1 FILTER NEWORKS AND NORMALIZER

The design of filter networks should meet the objective that they should accept a language as close to English as possible and they should be easily adaptable to a new DB; to meet both of them, we decided to use ATNs to be the filter networks i.e. a set of ATNs is defined to contain the grammar of the field descriptions. Normalizer, then, is an ATN interpreter. The ATN formalism we follow here is described in chapter 2.

All the actions of the ATNs are contained in a table called action-table. The action table is an assoc list with the following organisation.

(actions

(<ATN-name-1> <actions>)

(<ATN-name-2> <actions>)

.....

)

On successful completion of an ATN (as indicated by DONE arc in our formalism), the actions corresponding to that ATN are performed by the normalizer.

The ATNs used in this pass for describing th fields are given in appendix 1. The first two ATNs named S1 and E do not describe any fields but provide control on the normalization process. We consider them in detail below. Throughout this discussion, *s* means the query yet to be processed, *value* means the normalised query produced so far, and *w* means the current word.

The ATN S1 has two arcs; the first arc is a termination condition second is a self-loop. If the first succeeds, which means *s* is null and the input query is completely scanned, the normalizer goes to S1-1 and thereafter returns *value*. Otherwise, the second arc passes control to E which replaces first few words by their cannonical representation if they describe a field; otherwise, it treats the current word as the starting of no description and puts the word as it is in *value*. In

either case, control returns to the node S1 because of the self loop and then the above set of actions repeat.

The ATN corresponding to E checks if the current word *w* is the starting word of a field description.

Each arc of the ATN E is a call on another ATN describing a field. If any one of the calls is successful, i.e. if there is a field description starting with *w*, then the corresponding ATN returns the cannonical form of the field recognised. The ATN E on completion replaces the description with this cannonical form.

On the other hand if none of the arcs succeed, then the last arc "skip" is taken and it removes the current word from *s* and keeps it in the *value* thus treating it as starting of no description.

We have designed about 20 ATNs for 5 different fields. We shall show the considerations that went into the design of these ATNs by illustrating the design of the ATNs for one filter network.

Consider the following descriptions of department phrase.

"department of computer science" "dept of cs"

"dept = cs" "computer science dept"

Each of the above descriptions has two parts.

(i). The description which gives department name indicated by strings like "cs", "computer science".

(ii). The string of words having the tokens like "dept of", "department of", "dept = cs", etc

We design two ATNs to accept both the parts.

The ATN shown in Fig 4.1 will accept the name of a department consisting of two words. The first word is any one of the names "computer", "electrical", "chemical", etc. The second word is any one of the "science", "engineering", etc. Arc 1 accepts the first word if it is any of the words in the list bound to *departments* whereas the arc 2 accepts if the second word is any of "science", "engineering", etc.

To handle abbreviations we add a third arc as shown in Fig 4.2.

At the end of every ATN, a variable Y is bound to

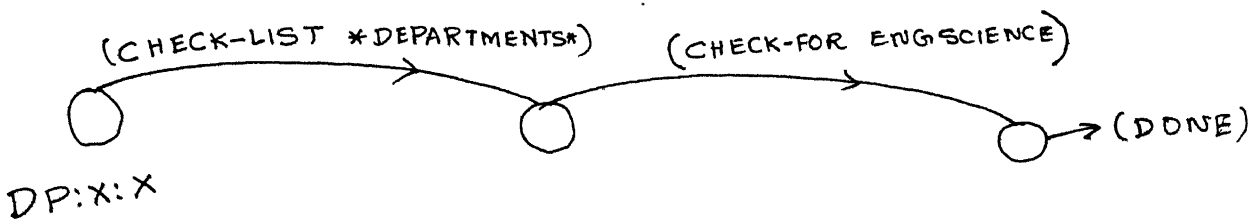


Fig. 4.1. AN ATN FOR DEPARTMENT NAMES.

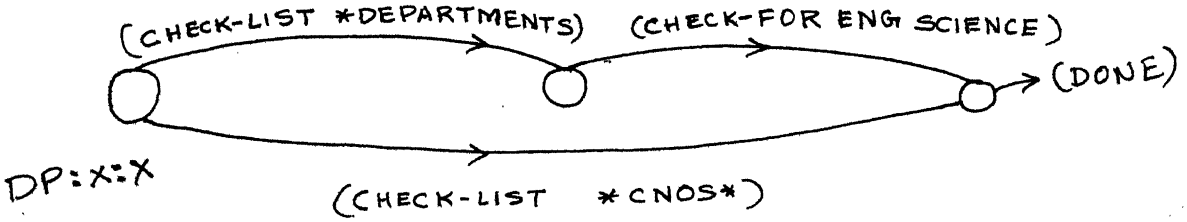


Fig 4.2. THE ATN INCLUDING ABBREVIATIONS

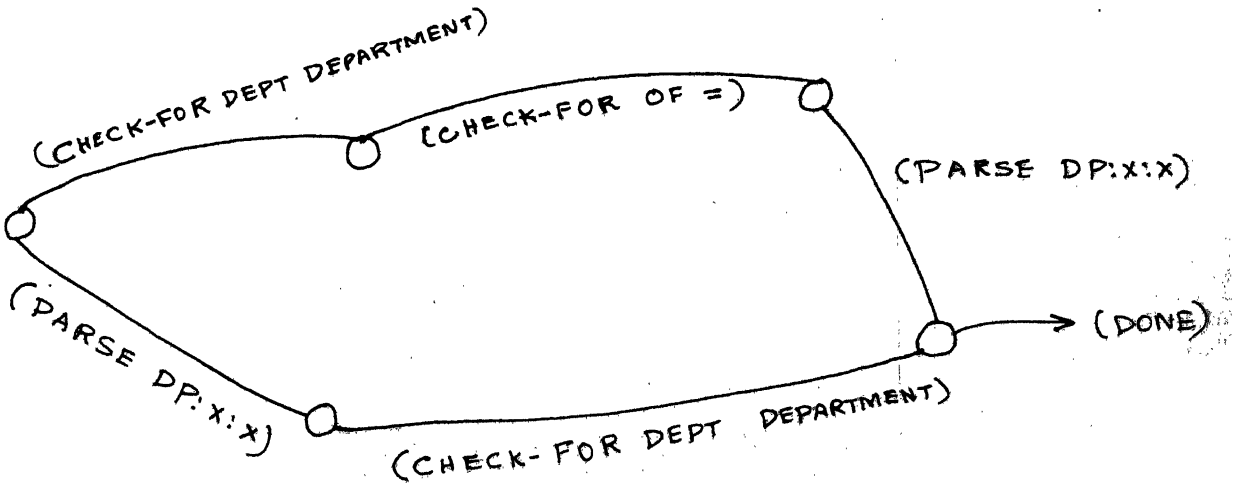


Fig 4.3. THE ATN FOR COMPLETE DEPARTMENT PHRASE

"(<name of the ATN> "syntactic structures that led to the success")"

So, we have to design the actions for this ATN in such a way that they take this Y and produce the formal output. By convention the output of an ATN kept as

"(<ATN-name> (field value))"

If "computer science" is recognised, then Y will be bound to (dp:x:x computer science). we want the pattern (dp:x:x (computer science)) to be returned as the result of the actions. To achieve such a change, the following program is written and kept in the action-table corresponding to this ATN.

```
(list 'dp:x:x (cdr y))
```

To design the second ATN to take care of part (ii) of department description we have the following specification:

(i). "dept" or "department" optionally followed by "of" or "=" and then necessarily followed by department name specification.

(ii). Department name optionally followed by "department" or "dept".

The ATN to recognise this is shown in Fig 4.3

The actions to be done on the successful recognition of this ATN should be such that they produce the cannonical form. At the end of the ATN, Y is bound to a list containing all the syntactic stuctures that led to the success of the ATN. For example, if the structure recognised is "dept of computer science", then Y will be bound to (dept-phrase dept of (dp:x:x (computer science))) But, we want a structure

(dept = <dept-name>)

So, to achieve this, we write a program as follows.

(list 'dept '= (cadr (assoc 'dp:x:x y)))

This program is kept in the action-table corresponding to this ATN.

Thus, we have seen how to design the ATNs for the "dept" field. The remaining ATNs can be designed similarly.

We conclude this section with an example.

"who taught cs450"

The first arc on the ATN S1 fails as *s* is not null. The next arc calls E. In the ATN E, the first five arcs fail because the starting word "who" does not satisfy any one of the ATNs describing the fields. So, the last arc "skip" takes "who" from *s* and keeps it in *value*. The same happens with the word "taught".

The next word "cs450" fails on "dept-phrase" arc and "name-phrase" of E. But in "course-phrase" it succeeds as follows. In the ATN corresponding to "course-phrase", the first arc "(parse cno:a)" is taken and control passes to the ATN cno:a.

There is only one arc in the ATN cno:a and this arc succeeds because all the three conditions on the arc are satisfied. This produces the structure "(cno:a cs450)". The control returns back to the ATN "course-phrase".

On return to the ATN "course-phrase", it succeeds and forms the structure "(cno = (cs450))" and control returns to the ATN E. In E the control goes to the node E1 and from there it goes to the top level ATN S1. The structure

returned by the ATN "course phrase" is substituted in the sentence. By now *s* has become null and the first arc on S1 succeeds and so control goes to S1-1 where it returns *value*. So we set the structure

"who taught cno = (cs450)"

4.3 PARSING

As disucssed in section 3.2.1, we decided to use syntactic grammar approach for our system. In this approach there are two stages. In this first stage the input query is taken and a parse is produced, while in the second stage the parse is the input and a formal query is the output.

We use ATN formalism for parsing (stage 1). The ATN formalism is discussed in detail in chapter 2. There are two basic parts in it.

(i). The lexicon and

(ii). The grammar and parser.

4.3.1 LEXICON

The constituents of the lexicon are the words used in the queries. The lexicon is logically devided into two parts, the core lexicon and the DB-specific lexicon. The core lexicon contains words which are DB-independent like "is", "are", "who" etc. The DB-specific lexicon contains words which are specific to the DB under consideration like "teach", "offer", etc.

As discussed in section 3.2.1, we keep in the lexicon the db-specific information of each word along with its lexical category. This is done to ease the process of local-table generation.

The logical structure of each entry of the lexicon is as follows.

(word (lex-info) (DB-specific info))

The lex-info gives the lexical category to which the particular word belongs. The following lexical categories are recognised.

(i). noun (ii). verb (iii). det (also known as article)

(iv). gerund (v). past-part(past participle)

(vi). prep (preposition)

(vii). int-pro (interogative pronoun) (viii). to-be

(ix). prop-noun (proper noun) (x). an-finite

(xi). ac-prep (xii). ps-prep (xiii). eq

Almost all of them are well known categories of English grammar. However, the following need a special mention.

(i). ac-prep : These are the prepositions which are used in active voice form of the verb as in

"who is teachins in the dept of cs"
—

"what was tausht to kumar"
—

(ii). ps-prep: It is exclusively the "by" used in the passive voice.

(iii). eq : it is any one of the three "=","<",">".

(iv). prep: any preposition, both ac-prep and ps-prep.

A word can belons to more than one catesory. We have to sive the list of catesories to which the word belonss. An example is shown below.

((to in of) (ac-prep prep) (no))

It shows that the words, to, in, and "of" belons to two catesories "ac-prep" and "ps-prep". "(no)" describes that there is no db-specific information associated with these words.

4.3.2 DB-SPECIFIC INFO

The nouns and verbs used in a particular DB may have special meanins. So, the fields which the nouns and verbs refer to are represented in a special slot called DB-info.

For nouns, the information is indicated by a single slot tagged "person" or "thins". Whether it is a person or a thins is decided by the int-pro used to refer to the noun.

The field referred to by the noun "course" is "cno". So, we keep the db-specific information of "course" as follows.

(cno (noun) (thins cno))

For verbs, the losical orsanisation is as follows.

(<word> (verb)
(doer (person ...) (thins ...))
(done (person ...) (thins ...))
)

doer: Those nouns which can act as subjects of the verb in active voice or objects of the verb in passive voice are represented in this slot.

For example, "A teacher offers..." or "... is offered by a teacher"

"A dept offeres ..." or "... is offered by a dept"

So, the doer slot of "offer" is

(doer (person tno) (thing dept))

Done! Those nouns which can act as the objects of the verb in active voice or subjects of the verb in passive voice are put in this slot.

For the same verb "offer", the done slot is as follos.

(done (person sno) (thing cno))

So, the complete representation of the verb "offer" is

```
(offer
      (verb)
            (doer (person tno) (thing dept ) )
            (done (person sno) (thing cno  ) )
)
```

4.3.3 PARSER AND GRAMMAR

The grammar is described by a set of ATNs and the driver routine is the ATN interpreter. The formalism is discussed in chapter 2.

The ATNs used in this phase are shown in appendix 2. We discuss the design of the ATNs in detail below.

The queries accepted by our grammar have a main clause(hence forth MC) followed by an arbitrary number ofsubordinate clauses(hence forth SC) and/or compound clauses (COMP-CL). The difference between subordinate clause and the compound clause is in their referents. The referent of subordinate clause is the last noun of the immediately preceding clause while the referent of a compound clause is the noun referred by any one of the previous clauses. The following query illustrates all the three types of the clauses.

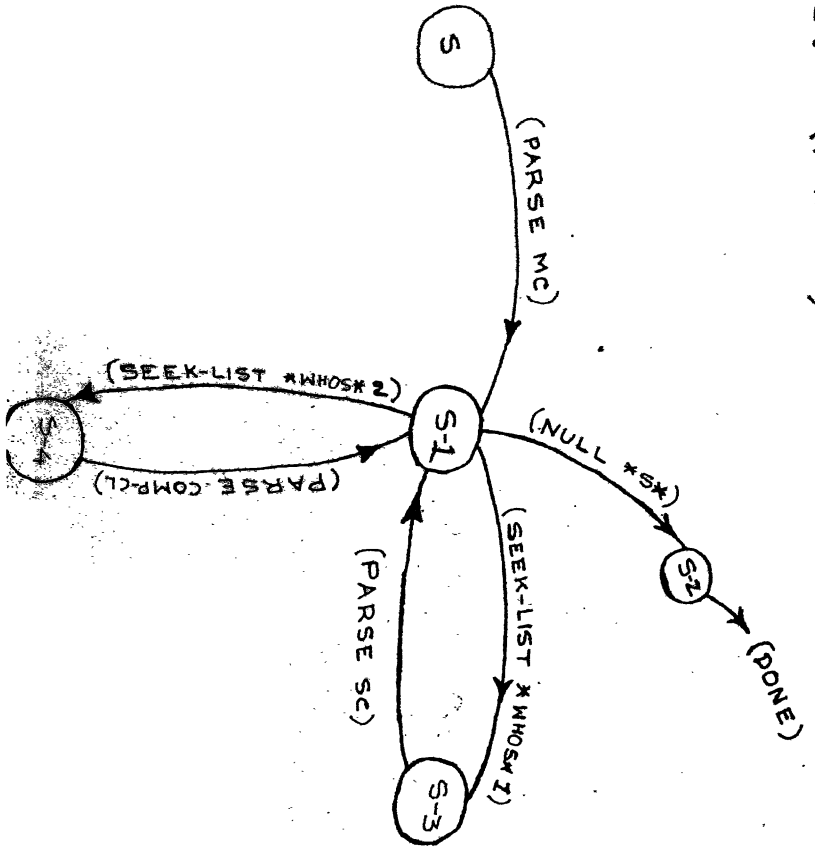
*who taught a course which is offered in the dept of ee
and who is in the cs dept

The main clause is "who taught a course", the subordinate clause is "which is offered in the dept of ee" and the compound clause is "and who is in the cs dept".

The structure of the query is MC followed by SC followed by COMP-CL. The SC refers to the last noun of the preceding clause (here "course") while, the COMP-CL refers to "teacher" which is refered by the MC. This major structure explains the top-level ATN S shown in Fig 4.4. The arcs from 1 to 3 and from 1 to 4 are to check whether the next clause is a SC or a COMP-CL. If the next clause is a sc, the first word of it is int-pro whereas if it is a COMP-CL int-pro should be in one of first two words. The arc from 1 to 2 is to check if the query is completely parsed.

Let us see the design considerations of MC. The ATN for MC is shown in Fig 4.4

Since our grammar accepts only interrogative queries, the first word of any query is an interrogative pronoun(henceforth int-pro). So, the first arc of MC accepts an int-pro as shown in Fig 4.4.



S: (P-ATN-1)

MAIN CLAUSE (P-ATN-2)

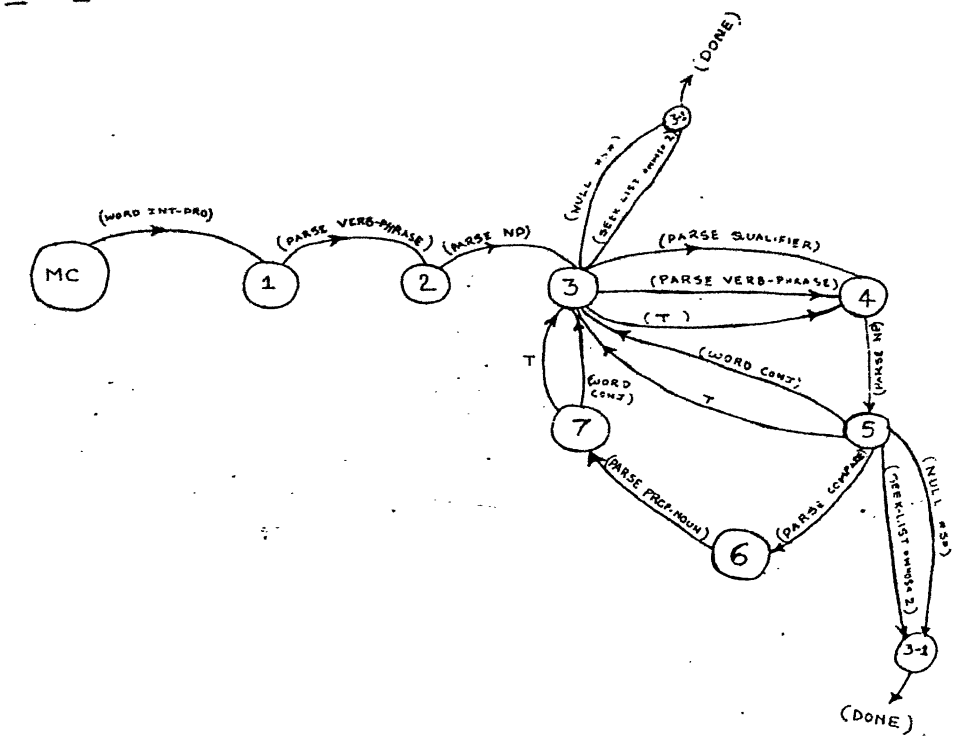


Fig 4.4 THE ATNS OF "S" and "MAINCLAUSE".

The next word of an interrogative query is a verb. The verb can be a simple to-be type of verb or it can be a passive or active voice of verbs like "teach", "take" etc. All of these combinations are recognised by the ATN verb-phrase. So, the next arc recognises this verb-phrase.

The next syntactic structure is the object of the verb in the previous syntactic structure. It can be a noun or an article followed by a noun. The ATN NP recognises these structures. So the next arc is NP.

These three arcs constitute the first part of MC. Also, these three arcs indicate what noun is referred to by the MC. We shall illustrate with examples.

"who teaches a course .."

"who is the teacher ...."

Thus, the verb phrase can implicitely contain the needed noun as in example 1 above or the needed noun can be explicitly given as in example 2 above ("teacher").

The design of the remaining part of MC is as follows.

In an interrogative query, we first indicate what we want and then provide other fields from which this neede field has to be found. The followins examples illustrate this.

"who is the teacher of cname = (data structures) "

"who is the teacher of the student of the dept offerins cno = ee424"

In the first example the needed field is "teacher" and the known field "cname " is given as an instantiated value. The connective between the noun "teacher" and the noun "cname" is the prep "of".

In the second example, the known fields are not instantiated, but are given as a chain of connectives followed by nouns.

Thus there are two basic ways in which the known fields can be given. The first one consists of series of connectives followed by noun phrases (as in "teacher of the student of the dept offerins cno..") whereas the second one consists of instantiations. Two parts are desisned in the ATN for MC to meet both these. The loop structure of the ATN in fis 4.4 shows these.

The connectives used in the first part can be a simple preposition as in "teacher of student.." or it can be a gerund as in "dept offering..." So, to meeet all these possibilities of a connective (or a qualifier) an ATN qualifier is designed. This recognises all the possible types of a qualifier. So the next arc between nodes 3 and 4 is this qualifier. The remaining arcs between the nodes 3 and 4 are to meet other possibilities of a qualifier. After this qualifier the next word is a noun. So, an arc to accept noun phrases is put between nodes 4 and 5. After this there are two branches corresponding to the two parts discussed earlier.

The first part namely the series of qualifier followed by noun phrases can be recognised by providing an arc back to node 3 from node 5 as shown. Thus structures shown in example 2 above are recognised by this loop "nodes 3 to 4, 4 to 5, 5 to 3".

The second part, namely the instantiation part, contains a noun followed by a relation operator followed by a proper noun. These two, the relation operator and proper noun are recognised by the two ATNs compare and np respectively. So, the next two arcs in the second branch

are from node 5 to 6 along compare and 6 to 7 along np and from 7 back to 3 along a jump or conj.

This finishes the design of mc. We shall illustrate with an example to show how various structures are recognised by MC.

"who taught a course in year = (84) in the dept = (computer science)"

The top-level ATN S is taken and the only one arc "(parse MC)" transfers control to the ATN MC. The various words in the above query parsed by MC are as follows

"who" : The first arc recognises this as a int-pro and the following structure is constructed.

(S (MC (INT-PRO (PERSON) WHO)))

"taught" : The second arc "(parse verb-ph)" recognises this as follows.

The ATN verb-ph is shown in Fig 4.5. The first arc "parse active-voice" takes control to the ATN "active-voice" shown in Fig 4.5. Since "taught" is not of the type to-be, the first arc fails. The next arc checks for verb. Since,

"taught" is a verb, it succeeds. So, this returns

"(ACTIVE-VOICE (VERB (DB-INFO) TAUGHT))" to the calling ATN namely "verb-phrase". This ATN checks if any prep is following it. Since the next word is not prep, it takes the jump arc succeeds, returning

"(VERB-PHRASE (ACTIVE-VOICE (VERB (DB-INFO) TAUGHT)))" to the calling ATN, namely the MC. The structure so far constructed is

```
"(MC (INT-PRO (PERSON) WHO )
     (VERB-PHRASE (ACTIVE-VOICE (VERB (DB-INFO) TAUGHT) ) )  )
```

"a course": The next arc is "parse NP". The ATN NP recognises the two words "a" and "course" as a noun phrase. So, the structure formed is

```
"(MC (INT-PRO (PERSON) WHO )
     (VERB-PHRASE (ACTIVE-VOICE (VERB (DB-INFO) TAUGHT)
     (NP (DET (NO) A) (NOUN (CNO) COURSE) )
  )
```

The control comes back to node 3 via the JUMP arc between node 5 and node 3.

"in" : The qualifier arc consumes this.

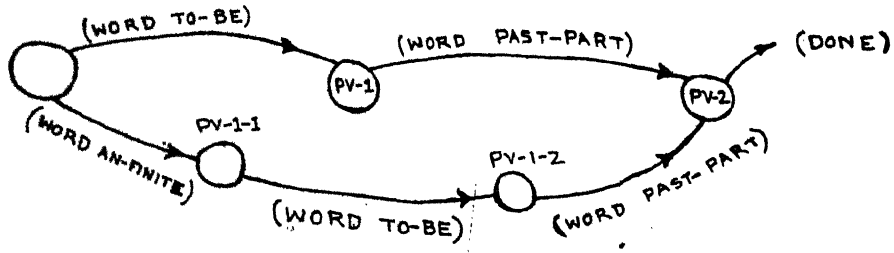
"year = (84)" : These three words are consumed successively by the three ATNs NP, COMPARE and NP by the three arcs from node 4 to 5, node 5 to 6, node 6 to 7.

The control comes back to the node 3 from the node 7 via the JUMP arc.

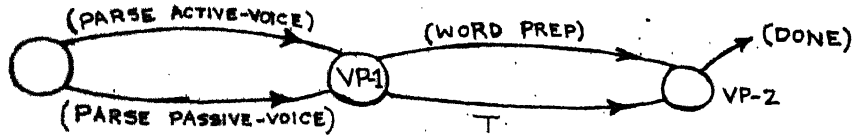
The remaining structures of the query are recognised similarly, returning the parse as follows.

```
(S
   (MC   (INT-PRO (PERSON) WHO)
         (VERB-PHRASE (ACTIVE-VOICE (VERB (DB-INFO)
                                          TAUGHT)))
         (NP (DET (DB-INFO) A)        (NOUN (CNO) COURSE) )
         (QUALIFIER (NO-INF (DB-INFO) IN)
         (NP (NOUN (YR) YEAR) )
         (COMPARE (EQ (DB-INFO) =)
         (PROP-NOUN (DB-INFO) (84) )
         (QUALIFIER (NO-INF (DB-INFO)  IN )
         (NP (NOUN (DEPT) DEPT) )
         (COMPARE (DB-INFO) = )
```

PASSIVE VOICE: (P-ATN-5)



VERB-PHRASE: (P-ATN-3)



ACTIVE-VOICE: (P-ATN-4)

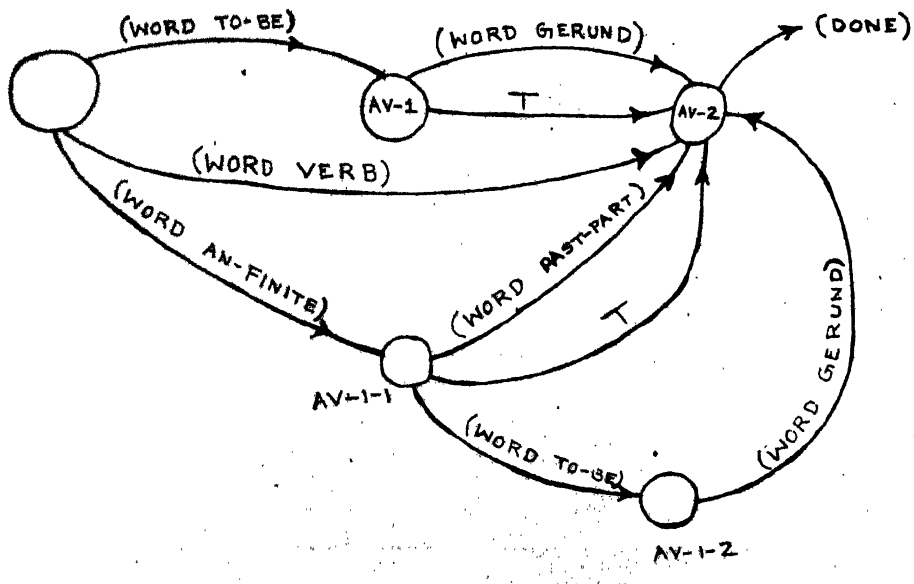


Fig 4.5 THE ATNS OF "VERB-PHRASE".

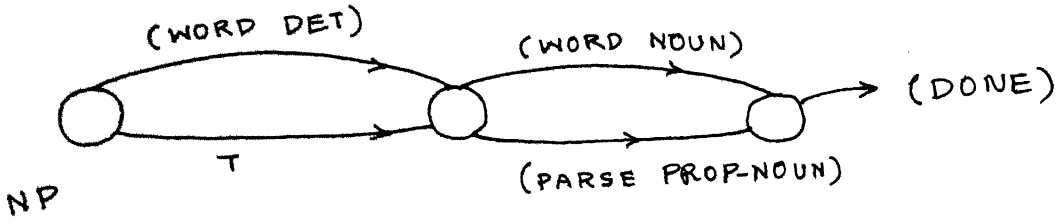


FIG. 4.5. ATN FOR NOUN PHRASE

```
        (PROP-NOUN (DB-INFO) (COMPUTER SCIENCE) )
    )
)
```

We shall see the design considerations of some of the remaining ATNs.

4.3.4 DESIGN OF NP

The ATN is shown in the Fig 4.5. A noun phrase can optionally have a determiner followed by a noun or a proper noun. The design of NP accounts for these cases.

4.3.5 DESIGN OF PROPER NOUN

The approach we followed is much more efficient when compared to the approach in PLANES and in LADDER.

In the LADDER for example, the names of the ships are a part of production rules and so, they are kept in the lexicon. We do not load the lexicon with the proper nouns. Instead, we use the followins approach.

The proper nouns are devided into two catesories.

(i). Those which can be recosnised at the normalization process.

(ii). Those which can be recosnised at parsins.

The first type of proper nouns are like "dept of cs", "data structures course", etc. Such proper nouns can be recosnised at the normalizer stase.

The second type of proper nouns occur in queries like : "who tausht kumar", "what was tausht by DRsansal", etc. Such proper nouns can be recosnised only by the context of the verb in which they are present On recosnisins such proper nouns the system confirms it by askins the user about it and then proceeds further. Whether such proper noun represents a teacher name or a student name is decided from the verb, the voice and whether it is a subject or a object.

Once a proper noun is defined, then if the same proper noun is used by the user in the next query, then the system proceeds without askins asain. This is achieved by keepins the user defined proper nouns temporarily in the lexicon.

4.3.6 VERB-PHRASE

The verb-phrase ATN is shown in Fig 4.5. A verb phrase can be a passive-voice followed by a preposition or a active-voice followed by a preposition. This accounts for the design of the ATN "verb-ph".

The ATN corresponding to active-voice is shown in Fig 4.5. The active voice can be a simple "to-be" or a "to-be" followed by a "gerund" or a simple "verb". These are shown in the ATN "active-voice" of Fig 4.5. The remaining arcs are used for the cases of perfect tense and perfect continuous. Including these arcs, we get the complete ATN "active-voice".

The design of the ATN "passive-voice" is similar.

The remaining ATNs are designed similarly.

CHAPTER 5

LOCAL-TABLE GENERATION

5.1 NEED FOR LOCAL-TABLE GENERATION

To generate a formal query, we must identify what fields are needed and what fields are required. But, a NL query, in general, does not give these fields explicitly. Instead, they are embedded in the nouns, verbs, and other syntactic structures. Local-table generator extracts these fields from the parse of the query and stores them in a data structure called 'local-table'. We illustrate it with an example.

"who teaches the student in the dept = (computer science)"

The word "student" implicitely conveys the field "sno". Similarly, the verb "teaches" conveys that the "tno" field is needed. Also, the query shows that "student" is to be found using the instantiated field "dept"; the "teacher" is to be found using the "student" thus obtained. Thus, there is a heirarchy along which the field has to be found. We call this "heirarchical tree". The subject of the query, referred to by "who", is "tno"; this is the final result

from the query. We call this field "Needed field". The field "sno", which is used to specify the path to find the needed field from the instantiated field is, called "specified field".

So, the entire information of the query can be illustrated as in the Fig 5.1

Local-table generator extracts such information from the query and stores it in the local-table.

5.2 DEFINITIONS OF TERMS USED

We define certain terms that are used in further discussion. We use the example shown in Fig 5.2 as reference.

"who is the teacher of the student in the dept = (computer science) taking a course which is offered by the Tname = DrRsansal".

Fig 5.2. An example query.

The query shown above has two clauses and the parser produces a parse for each of them separately. The first clause is MC and the second is SC. The following terms are defined for each clause.

NEED : TNO ; HEIRARCHY :

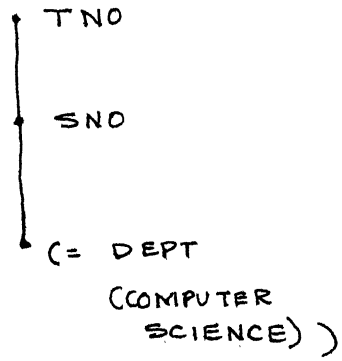


Fig. 5.1. THE NEEDED FIELD AND HEIRARCHY OF "who Teaches the student in the dept of CS"

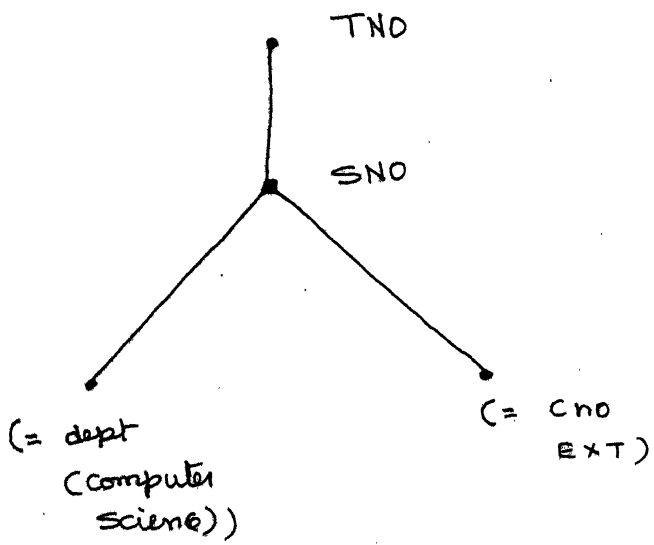


Fig 5.3. THE HEIRARCHICAL TREE OF MC OF Fig 5.2.

5.2.1 NEEDED FIELD

Each clause has a subject which is referred to by its interrogative pronoun. The field corresponding to the subject is called the "needed field".

In the first clause, "tno" (corresponding to the word "teacher") is referred to by the interrogative pronoun "who". So, "tno" is needed field of MC. The second clause refers to "cno"(corresponding to the word "course") and so, "cno" is its Needed field.

5.2.2 INSTANTIATED FIELD

A clause may instantiate some fields to some values. These fields are called instantiated fields.

In the first clause the field "dept" is instantiated to "(computer science)" and so, "dept" is called instantiated field of the MC. Similarly "tname" is instantiated field of SC.

5.2.3 EXTERNAL FIELD

In a clause, a field may only be mentioned, but the properties it satisfies are given in a later clause. Such a field is called external field as its description is

external to the clause in which it is present.

In the example of Fig 5.2 "cno" (the word "course") is mentioned in MC but the way of finding is given in SC. So, it is called external field in MC.

5.2.4 PATH FIELD

Each clause may use some fields to specify the path of finding the needed field from the instantiated fields. These fields, used for specifying the path, are called path fields.

In the above example, MC uses "student" as an intermediate field to get the needed field "tno" from both the external field "cno" and the instantiated field "dept". Actually, it specifies that "sno" has to be found from the "dept" and "cno" (which is described by a latter clause) and the "sno" so obtained must be used to get "tno". So, "sno" is used for specifying the path and hence, it is called path field (of MC).

For SC there is no path field. The needed field has to be obtained directly from the instantiated field.

5.2.5 HEIRARCHY

There is a heirarchy on various fields in a clause. The heirarchy indicates how each field is to be found from the others.

In the example above, MC indicates that

(i). "sno" has to be found from "dept" and "cno".

(ii). "tno" has to be found from the "sno" determined above.

Thus there is a heirarchy in the fields; this heirarchy is mapped into a tree data structure and stored in the local-table. The way of mapping is as follows.

"If f1 is to be determine using the fields f2,f3 then, the fields f2,f3 must be put as sons of f1."

So, the needed field is put as the root of the tree (henceforth "heirarchical tree"). Those fields which must be used to determine it are made as its sons. Each of the son has, as its sons, the fields which determine it and so on. This process is repeated. The leaf nodes are either instantiated nodes or the external nodes.

For our example MC, the needed field is "tno" and so, it is put as root. The field "sno" should be used to find "tno", so it is put as son of the root node "tno". The

"tno" has to be found from the instantiated field "dept" and the external field "cno". So, they are kept as sons of this node. The final tree is as shown in Fig 5.3.

5.3 LOCAL-TABLE STRUCTURE

A local-table is a data structure in which the information extracted from a clause is stored. A query has as many local-tables as its clauses.

A local-table has four slots.

5.3.1 TYPE SLOT

The first slot gives the type of the clause for which the local-table is generated. We know from the previous chapter that there are three types of clauses, MC, SC, and COMP-CL. So, this slot contains information regarding this.

5.3.2 NEED SLOT

This slot contains the needed field of the clause. So, it is called the need slot.

5.3.3 VARS SLOT

This slot contains the heirarchical tree of the clause.

5.3.4 EXT SLOT

This slot contains the external field of the clause, if any.

The local-tables of the example of Fig 5.2 are shown in Fig 5.4.

5.4 FIELD EXTRACTION

Our aim is to design an algorithm which takes the parse of a query and returns a local-table.

There are basically two ATNs which accept the queries. One is the ATN corresponding to MC and the other SC. We explain the design of the algorithm for extracting fields from the MC and the process of extracting the fields from SC is similar.

The ATN corresponding to MC is shown in Fig 5.5. Semantically, it can be devided into two parts. The first part encapsulates the different ways of expressing the needed field of a clause. The second part indicates the different heirarchical structures possible among various

```
( (LOCAL-TABLE
        (TYPE  MC)
        (NEED  TNO)
        (VARS
            (TNO
              (SNO
                (= DEPT (COMPUTER
                         SCIENCE))
                (= CNO 'EXT))
            )
        )
    )
    (LOCAL- TABLE
        (TYPE SC)
        (NEED  CNO)
        (VARS
            (CNO
              (= TNAME
                 DRRSANGAL)
            )
        )
    )
)
```

Fig. 5.4. LOCAL-TABLES OF Fig 5.2.

fields. The first part consists of arcs of nodes : 0 to 1; 1 to 2; 2 to 3 of the Fig 5.5. The second part consists of the remaining loop like structure.

The first part is designed to accept any needed field, either explicitely as a noun or implicitely in a verb, either in passive voice or active voice. So, the design of an algorithm to extract needed field is based on these arcs. It is discussed in section 5.4.1.

The second part is used to describe the heirarchy of one field over the other. So, this part is used to design an algorithm for extracting the heirarchical-tree. This is discussed in section 5.4.2.

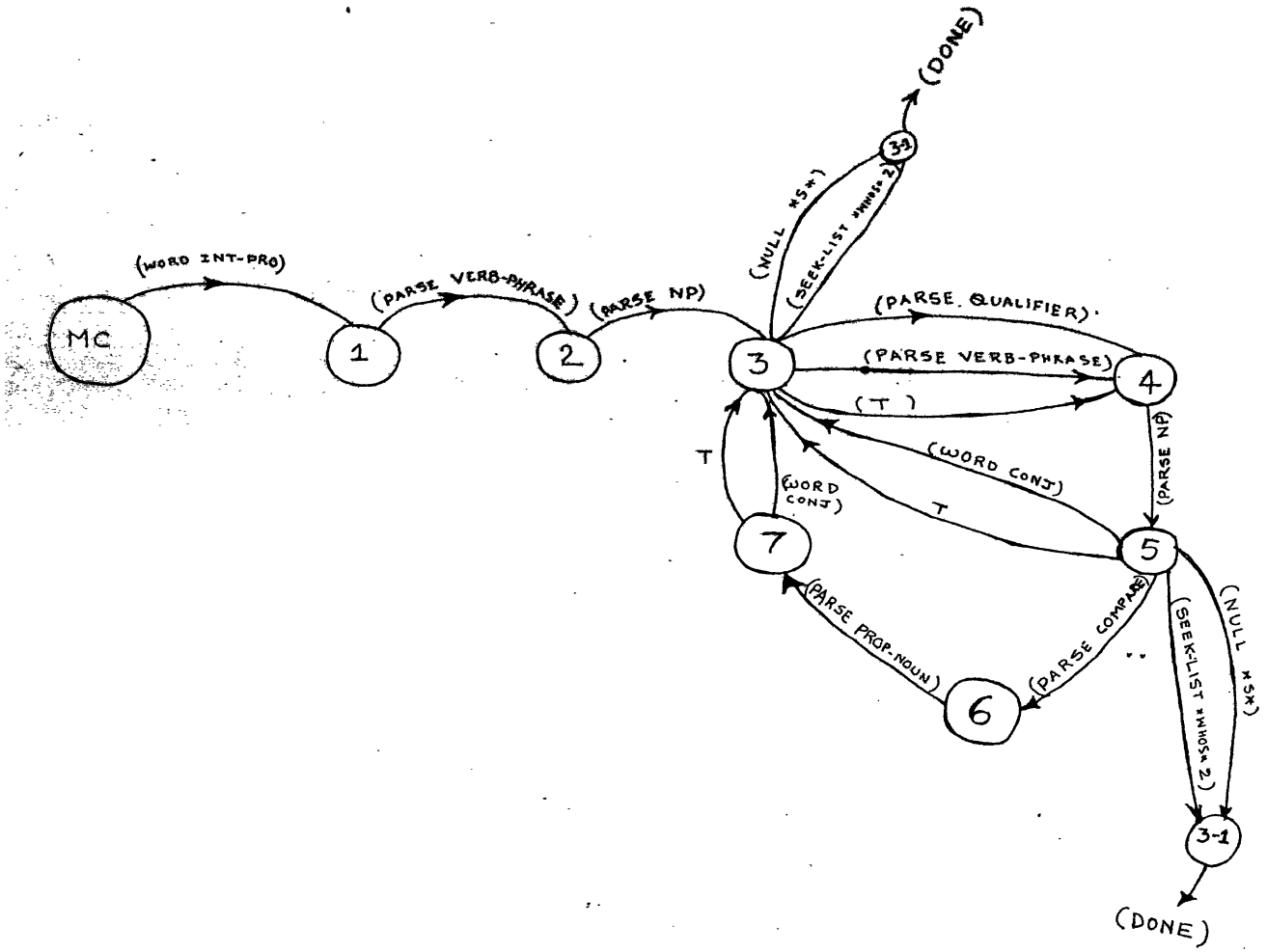
5.4.1 EXTRACTION OF NEED FIELD:

The needed field of any clause can be extracted from the first three syntactic structures corresponding to the first three arcs of the ATN Fig 5.5.

There are various ways in which a needed field can be there in a clause. They are shown below.

(1). "who is the teacher of cname = (data structures)"

MAIN CLAUSE (P-ATN-2)



(2). "who taught cname = (data structures)"

(3). "who was taught in the dept = (computer science)"

(4). "who is teaching cname = (data structures)"

The parses of these queries are shown in the Fig 5.6

In the first case, (example 1 above) the verb in the verb-phrase has only "to-be" type verb which carries no DB-specific information. The needed field is in the noun of the next noun-phrase (indicated as NP in the parse). Hence for such types of queries, which have "to-be" type of verbs as the main verbs, we can extract the needed field from the noun of the NP following the verb-phrase.

For the example 1, the needed field has to be obtained from the noun "teacher". The field referred to by this noun is in the parse beside the noun itself. This is contained by the <db-specific info> slot of the noun. In this example the field referred to by "teacher" is "tno".

Thus, when the main verb is of the type "to-be" then the needed field is to be taken from the noun phrase represented by the third arc of Fig 5.5. The needed field referred by the noun of the noun phrase is available in the parse beside noun.

The rest of the examples have the needed field implicitely mentioned in the verb or gerund. To extract the needed field from the verb, we have to know what the verb refers to. So, we shall see what the <DB-specific-info> slot of a verb contains and then decide how to extract the field referred to by the verb.

The <DB-specific-info> slot of any verb has two main slots. One is "doer" slot and the second is "done" slot. The "doer" slot indicates the subjects of the verb while the "done" slot indicates the objects of the verb.

The <DB-specific-info> slot of the verb "offer" is as shown in the Fig 5.7.

In the Fig 5.7, the "doer" of "offer" can be a teacher or a department. So, the fields corresponding to the two nouns are put in the "doer" slot. The slot named "person" indicates that the field in that slot is of "person" type. Similarly the non-person type of fields are put in the slot "things". In the above example, in the "doer" slot, the field "tno" is of "person" type and so, we keep this in the "person" slot; the field "dept" is of non-person type of slot, and so, we put it in the "things" slot. The same rule is followed in the "done" slot also.

```
(DOER (PERSON TNO) (THING DEPT))
(DONE (PERSON SNO) (THING CNO) )
)
```

Fig 5.7. DB-SPECIFIC INFORMATION OF "OFFER"

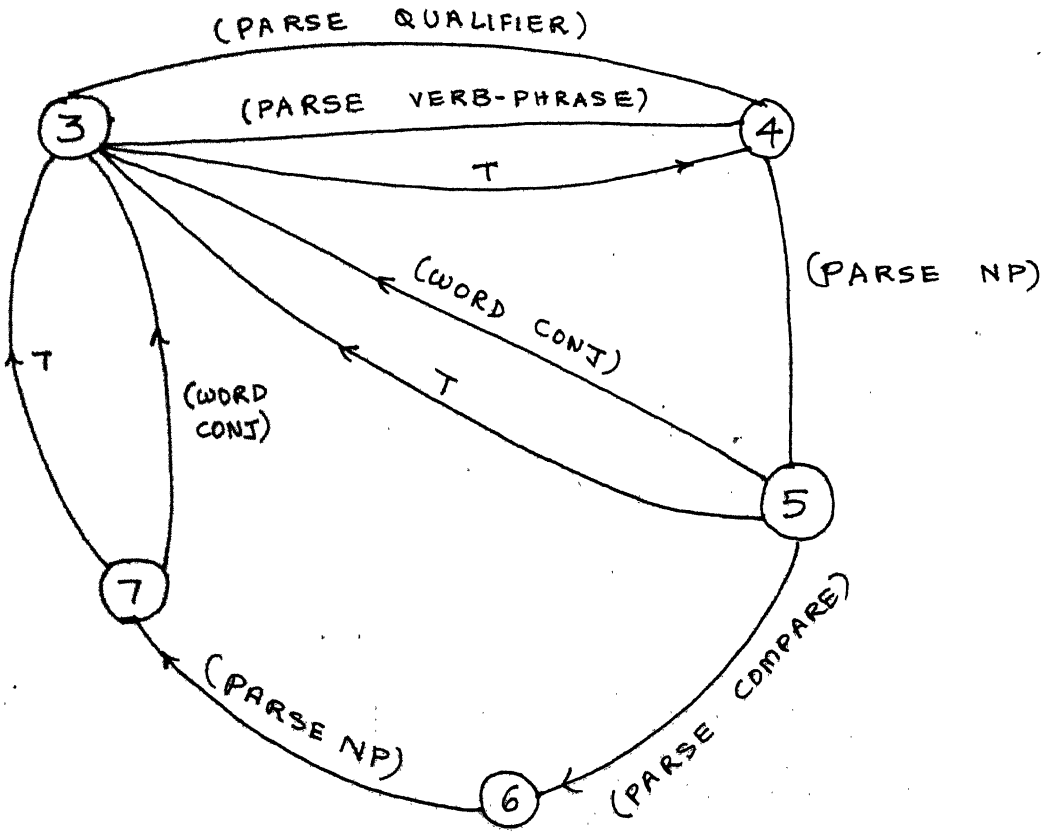


Fig 5.8 LOOP LIKE STRUCTURE ENCAPSULATING THE HEIRARCHICAL TREE.

We shall see how to use this representation to extract needed field. If the verb phrase is in active voice, as indicated by the parse, then the needed field is the subject of the verb. So, the needed field is taken from the "doer" slot of the <db-specific-info>. Whether we have to take the field in the "person" slot or "thing" slot is decided according as the "int-pro" is a "person" type(like "who") or a "thing" type (like "which" "what").

Similarly, the "done" slot is taken when the "verb-phrase" is in passive voice. So, the rule to extract the needed field from the verb is as follows.

"Decide the major slot "doer" or "done" according as the "verb phrase" is in active voice or in passive voice (this is indicated by the parse). The lower slot "person" or "thing" is decided according as the "int-pro" is a "person" or "thing" type (this is available in the parse beside the word corresponding to "int-pro")."

So, by using this rule, the needed fields of the examples 2,3,4,5 are respectively, "tno", "sno", "cno", and "tno"; the <db-specific-info> of the verb "teach" is same as that of "offer" (shown in Fig 5.7).

After finding the needed field of a clause the corresponding slot in the local-table of the clause is filled with this field. Also, the needed field constitutes

the root node of the heirarchical tree of the clause.

The part of the ATN used for finding the needed field is the first three arcs. The rest of the ATN is used to decide the remaining part of the heirarchical tree. This is discussed in the following section.

5.4.2 EXTRACTION OF HEIRARCHICAL TREE

To build up the heirarchical tree, we need to recognise the fields referred to by the noun phrases (indicated as NP in parse) and put them in proper structure. The part of ATN used for this purpose is shown in the Fig 5.8. This has basically two parts. Each part recognises specific sentence fragments with specific semantics.

The first part recognises qualifier phrases. Nodes 3 to 4, 4 to 5, 5 to 3 constitute this part. A qualifier phrase consists of a qualifier followed by a NP. The following example shows it.

"who is the teacher of the course in the dept of the student".

The parse of the above clause is shown in the Fig 5.9. The parse shows that it consists of sequence of "QUALIFIER" followed by "NP".

```
(VERB-PHRASE (ACTIVE-VOICE (TO-BE (NO-INF) IS)))
(NP (DET (NO-INF) THE) (NOUN (TNO) TEACHER))
(QUALIFIER (NO-INFO (PREP (NO-INF) OF)))
(NP (DET (NO-INF) THE) (NOUN (CNO) COURSE))
(QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
(NP (DET (NO-INF) THE) (NOUN (DEPT) DEPT))
(QUALIFIER (NO-INFO (PREP (NO-INF) OF)))
(NP (DET (NO-INF) THE) (NOUN (SNO) STUDENT))
)
```

Fig 5.9. PARSE OF THE MAIN CLAUSE

"who is the teacher of the course in the dept of the student"

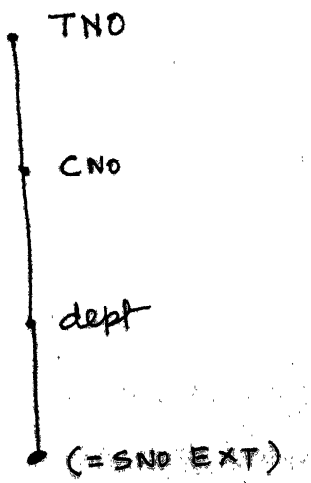


Fig. 5.10. HEIRARCHY OF Fig 5.9.

The first three structures determine the needed field. The syntactic structures that follow are a series of qualifier followed by noun phrase.i.e a sequence of qualifier phrases. The semantics of such qualifier phrases is that the noun of each qualifier phrase determines the noun of the following the qualifier phrase. So, we extract the fields represented by each qualifier phrase and put them in the heirarchical tree such that the tree reflects this semantics.

In the example, "course" determines the "teacher"; so, "cno"(the field corresponding to course) is the son of "tno". Similarly, "dept" is son of "cno". The heirarchical structure is shown in Fig 5.10

The second part recognises instantiated nouns. Nodes 3 to 4, 4 to 5, 5 to 6, 6 to 7, and 7 to 3 constitute this part. An instantiated noun indicates that a noun is instantiated to a value. The following example illustrates this.

"who is teaching cname = (data structures) in the dept = (computer science) and in the year = (84)".

In the above example the fields "cname","dept", and "year" are instantiated respectively to "(data structures)", "(computer science)", and "(84)". The parse of the above query is shown in the Fig 5.11.

```
(MC (INT-PRO (PERSON) WHO)
    (VERB-PHRASE (ACTIVE-VOICE (TO-BE (NO-INF) IS)
                               (GERUND ((DOER (PERSON TNO)
                                              (THING DEPT))
                                        (DONE (PERSON SNO)
                                              (THING CNO))
                                        )
                                       TEACHING)) )
    (NP (NOUN (CNAME) CNAME))
    (COMPARE (EQ (NO-INF) =) )
    (NP (PROP-NOUN (FIELD VALUE) (DATASTRUCTURES)))
    (QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
    (NP (DET (NO-INF) THE) (NOUN (DEPT) DEPT))
    (COMPARE (EQ (NO-INF) =) )
    (NP (PROP-NOUN (FIELD VALUE) (COMPUTERSCIENCE)))
    (CONJ (NO-INF) AND)
    (QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
    (NP (DET (NO-INF) THE) (NOUN (YR) YEAR))
    (COMPARE (EQ (NO-INF) =) )
    (NP (PROP-NOUN (FIELD VALUE) (84)) )
)
```

Fig 5.11. THE PARSE OF

"who is teaching cname = (data structures)
in the dept of (Computer Science)
and in the year = (84)".

This part recognises the strings like "noun phrase", "compare", and "noun phrase(containing a proper noun)". This string instantiates the noun in the noun phrase to the value indicated by the proper noun of the second noun phrase.

To build the heirarchical tree for the query shown in Fig 5.11 we to add, as children, all the nodes corresponding to the instantiations. In the above example, "tno" is to be determined from the five instantiated fields. The semantics is maintained in the heirarchical tree by putting all the instantiated fields as sons of the needed field. The heirarchical tree of the above query is shown in Fig 5.12

Thus we see that instantiated noun structures do not contribute to the depth of the tree, instead, they contribute to the breadth.

A practical example consists of both the parts as shown below.

"who is the teacher of the course in year = 84 offered in the dep of sname = kumar"

The extraction of heirarchical structure from such queries follows both the methods described above. This is described in the following section.

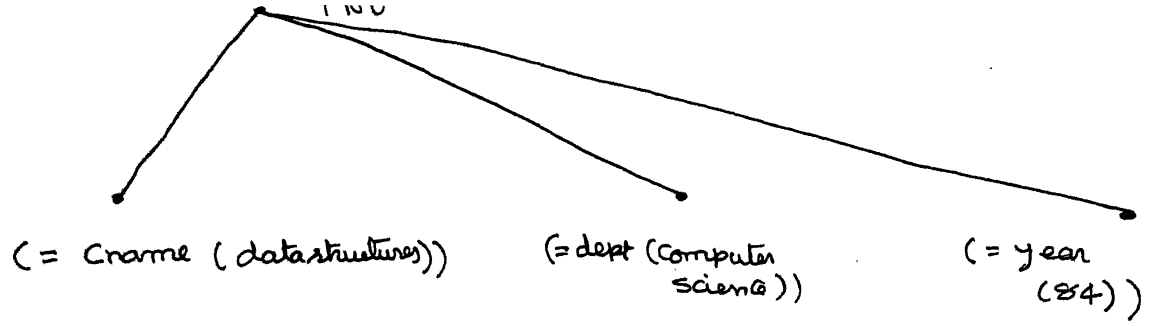


Fig 5.12. HEIRARCHICAL TREE OF Fig 5.11.

```
(MC (INT-PRO (PERSON) WHO) (VERB-PHRASE (ACTIVE-VOICE
                                          (TO-BE
                                            (NO-INF) IS)))

    (NP (DET (NO-INF) THE) (NOUN (TNO) TEACHER))
    (QUALIFIER (NO-INFO(PREP (NO-INF) OF)))
    (NP (DET (NO-INF) THE) (NOUN (CNO) COURSE))
    (QUALIFIER (NO-INF (PREP (NO-INF) IN)))
    (NP (NOUN (YR) YEAR)) (COMPARE (EQ (NO-INF)
                                       =))
    (NP (PROPER-NOUN (FIELD-VALUE) (83)))
    (QUALIFIER ( AC-QUA ( PAST-PART (< DB-INFO>)
                                    OFFERED)
                        (PREP (NO-INF) IN)))
    (NP (DET (NO-INF) THE) (NOUN (DEPT) DEPT))
    (QUALIFIER (NO-INF (PREP (NO-INF) OF)))
    (NP (DET (NO-INF) THE) (NOUN (SNAME) SNAME))
    (COMPARE (EQ (NO-INF) =))
    (NP (PROP-NOUN (FIELD-VALUE) Kumar))    )
```

Fig. 5.13. PARSEOP "who is the teacher of the course in year = (83) offered in the dept of Sname = Kumar".

5.4.3 COMPLETE EXAMPLE

We shall illustrate both the above steps by extracting the needed fields and the heirarchical tree for a typical query.

The example we consider is shown below.

"who is the teacher of the course in year = 83 offered in the dept of the sname = kumar"

The parse is shown in the Fig 5.13

The first three structures of the parse decide the needed field. As explained in the section 5.4.1, the needed field has to be found from the noun of the NP following the verb phrase. This is found from the parse to be "tno".

This needed field is kept as root node of the heirarchical tree. The remaining query from which the fields are extracted is

"of the course in the year ....."

The first noun "course" belongs to the first type of structure, namely, the qualifier phrase type of structure. So the field represented by "course" is added as a son of the latest non leaf node of the heirarchical node. So, the heirarchical tree is now as shown in Fig 5.14.

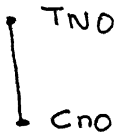


Fig 5.14. PART OF THE HEIRARCHICAL TREE OF Fig 5.13.

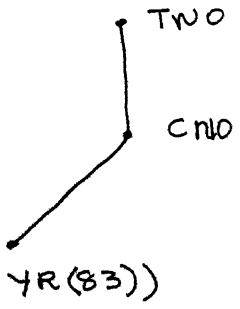


Fig 5.15 PART OF THE HEIRARCHICAL TREE OF Fig 5.13

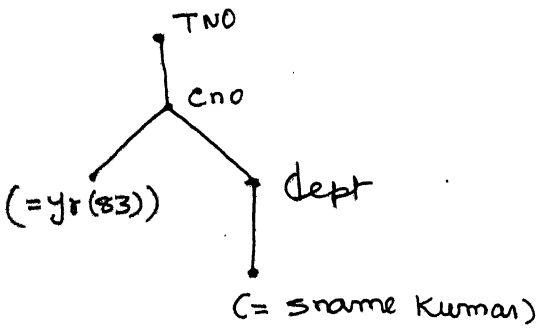


Fig 5.16. COMPLETE HEIRARCHICAL TREE OF Fig 5.13.

Now the query to be processed still is

"year = 84 offered in ...."

The noun "year" is a part of the instantiation. So, it is attached to a child of the previous non-leaf node, but we remeber that latest non-leaf node is still "cno" and any fields found further are to be attached to "cno" but not to "year". The field represented by the noun "year" is found from the parse and the node indicating the instantiation is added to the above formed heirarchical tree Which is now as shown in Fig 5.15.

The remaining query consists of a qualifier phrase followed by the instantiation. So, the final tree structure constructed is as shown in Fig 5.16.

5.5 DEREFERENCING AND DESCOPING

For queries consisting of more than one clause, which is common, the SC and the COMP-CL refer to nouns of preceding clauses. To determine the needed field of each of these clauses(namely SC and COMP-CL) we must identify the referents of these clauses. This is what we call dereferencing.

Also, when more than one clause is there, it is quite likely that, a field is mentioned as external in one clause but, the way of finding is specified in a later clause. We eliminate this forward reference to facilitate the query processor (which takes the formal query from our system and interacts with the DB). This is achieved by reordering the local-tables. Another reason for reordering the local-table structure is to identify those clauses which together determine a field and to group those clauses together. This process of reordering the local-tables is called descoping.

We shall illustrate with an example.

"who taught a course which is offered in dept = (computer science) and which is offered in year = (84) and who is in dept = (electrical engineering)".

The query contains 4 clauses. The first one is MC. The second one is SC. The third and fourth are COMP-CLs. The parse of the query is shown in the Fig 5.17

The structure of the above query (in terms of clauses) can be put as below.

MC SC COMP-CL1 COMO-CL2

```
(S (MC (INT-PRO (PERSON) WHO)
       (VERB-PHRASE (ACTIVE-VOICE (VERB
                                   ((DOER (PERSON TNO)
                                          (THING DEPT))
                                    (DONE (PERSON SNO)
                                          (THING CNO)))
                                   TAUGHT)))
       (NP (DET (NO-INF) A) (NOUN (CNO) COURSE)))
   (SC (INT-PRO (THING) WHICH)
       (VERB-PHRASE
                (PASSIVE-VOICE
                 (TO-BE (NO-INF) IS)
                 (PAST-PART (<db-info>) OFFERED)))
       (QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
       (NP (NOUN (dept) DEPT)) (COMPARE (EQ (NO-INF) =))
       (PROP-NOUN (FIELDVALUE) (COMPUTER SCIENCE)))
   (COMP-CL (CONJ (NO-INF) AND)
            (INT-PRO (THING) WHICH)
            (VERB-PHRASE (PASSIVE-VOICE
                          (TO-BE (NO-INF) IS)
                          (PAST-PART (<db-info>)
                                     OFFERED)))
            (QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
            (NP (NOUN (YR) YEAR)) (COMPARE (EQ (NO-INF)
                                               =))
            (PROP-NOUN (FIELDVALUE) (84)))
!  (COMP-CL (CONJ (NO-INF) AND) (INT-PRO (PERSON) WHO)
           (VERB-PHRASE (ACTIVE-VOICE (TO-BE (NO-INF) IS)))
           (QUALIFIER (NO-INFO (PREP (NO-INF) IN)))
           (NP (NOUN (DEPT) DEPT)) (COMPARE (EQ
                                             (NO-INF)
                                             =))
           (PROPER-NOUN (FIELDVALUE) (ELECTRICAL ENGINEERING)
   )
```

Fig. 5.17. THE PARSE OF A TYPICAL QUERY.

The needed field of SC can be found from the fact that an SC always refers to the last noun of the immediately preceding clause. So, the referent of the SC is "course" (the corresponding field being "cno"). The referents of comp-cl1 and comp-cl2 are to be found by looking backwards at the needed field of each clause. The referent of a comp-cl is the needed field of the nearest clause whose needed field could fit to be the needed field of the comp-cl. In the above example the referent of comp-cl1 is the needed field of sc rather than the needed field of MC because, the needed field of MC, the field "tno", does not fit to be the needed field of COMP-CL1 (the string "which is offered.." in comp-cl1 suggests that the referent is a course rather than a teacher). Similarly, the needed field of COMP-CL2 is found to be "tno". After dereferncing, that is, after the needed fields of the clauses are found, the heirarchical structure is extracted and the local-tables are formed. The local-tables are as shown in Fig 5.18.

Thus, the rules to be followed to dereference are as follows.

For an SC, the last noun of the immediately preceding clause gives the referent.

The COMP-CLs are dereferenced by using standard block-structure referencing rules. This is done as follows. Since a COMP-CL refers to a noun already referred by another clause, to dereference a COMP-CL, we start looking backwards. The needed field of the nearest clause which also fits to be the need field of the present COMP-CL is taken the referent of the COMP-CL. Whether it fits or not is decided from the interrogative pronoun and verb phrase of the COMP-CL.

5.5.1 DESCOPING

In the local-tables of Fig 5.18, the second and third local-tables together determine the "cno" while, the first and fourth local-tables together determine the "tno" field. Also the first local-table has an external field "cno", finding of which is specified in the next two local-tables. corresponding to SC and COMP-CL1. So, to achive both, the elimination of the forward reference and and to group together the local-tables which describe essentially the same fields, we do the descoping.

The first step to be done in descoping, is to group together those local-tables which are the innermost and put them at the beginning of the reorder-list. By innermost we mean that those local-tables which have no external

```
( (LOCAL-TABLE (TYPE MC)
               (NEED TNO)
               (VARS (TNO (= CNO EXT)))
               (EXT CNO))

 (LOCAL-TABLE (TYPE SC) (NEED CNO)
              (VARS (CNO (= DEPT (COMPUTER SCIENCE))))
              (EXT NIL))

 (LOCAL-TABLE (TYPE COMP-CL) (NEED CNO)
              (VARS (CNO (= YR (84))))
              (EXT NIL))

 (LOCAL-TABLE (TYPE COMP-CL) (NEED TNO)
              (VARS (TNO (= DEPT (ELECTRICAL ENGG))))
              (EXT NIL)) )
```

Fig 5.18. LOCAL-TABLES OF THE EXAMPLE QUERY

```
( [(LOCAL-TABLE (TYPE SC) (NEED CNO)
                (VARS (CNO (= DEPT (COMPUTER SCIENCE))))
                (EXT NIL))

   (LOCAL-TABLE (TYPE COMP-CL) (NEED CNO)
                (VARS (CNO (= YR (84))))
                (EXT NIL)) ]

  [ (LOCAL-TABLE (TYPE MC) (NEED TNO)
                 (VARS (TNO (= CNO EXT)))
                 (EXT CNO))

    (LOCAL-TABLE (TYPE COMP-CL) (NEED TNO)
                 (VARS (TNO (= DEPT (ELEC ENGG))))
                 (EXT NIL)) )
```

Fig. 5.19. LOCAL-TABLES AFTER REORDERING

reference and which together determine the same field.

In the above example we see that second and third are the innermost group of local-tables because they have no external fields and they together determine the same need field "cno". So, we keep these two local-tables together.

The second step of descoping is to find the next higher group of clauses and to place them beneath the previous group in the reorder-list. By "next higher" we mean those group of clauses which refer to the same field and which have external fields, if any, determined by the previous groups of clauses. This step is applied until all the local-tables are exhausted.

In the above example we see that MC and COMP-CL2 have the same needed field and also their external fields (actually, only MC has an external field "cno" which is determined by SC) determined by the previous group of local-tables. So, we keep this group together and beneath the previous group. With this reordering, the local-tables are as shown in the Fig 5.19.

The final step is to put each group of local-tables in a single list tagged with "intersect". This is done to indicate the fact that the answer is found by intersection of the answers of both the local-tables. In the above example, in Fig 5.19, the first two clauses together

```
                    (VARS (CNO (= DEPT
                                  (COMPUTER
                                    SCIENCE)))
                    (EXT NIL))

          (LOCAL-TABLE (TYPE COMP-CL)
                       (NEED CNO)
                       (VARS (CNO (=YR (84))))
                       (EXT NIL))]

[INTERSECT (LOCAL-TABLE (TYPE MC)
                        (NEED TNO)
                        (VARS (TNO (= CNO EXT)))
                        (EXT CNO))

           (LOCAL-TABLE (TYPE COMP-CL)
                        (NEED TNO)
                        (VARS (TNO (= DEPT
                                      (ELECTRICAL
                                        ENGG))))
                        (EXT NIL))])
```

Fig 5.20. LOCAL-TABLES AFTER DESCOPING IS COMPLETE.

determine the "cno". Thus, "cno" to be found must satisfy the first local-table as well as the second local-table. So, we indicate this by keeping them in a single entity tagged with "intersect". Similarly the third and fourth are tagged. The final form of local-tables are shown in Fig 5.20

CHAPTER 6

SEMANTIC GRAPH DRIVER AND FORMAL QUERY GENERATION

6.1 INTRODUCTION

The output of the local-table generator gives the fields needed, the fields given and the heirarchy existing among the various fields. This output does not contain such information as what relations are to be chosen, what operations like selection, projection, and join are to be performed on the identified fields.

The semantic graph driver gives such information in the form of a Formal Query. This process of generation of formal query is dependent on the relational structure, linking information and the key fields of the relations of the data base.. This information of a DB is abstracted into a graph which we call Semantic Graph and the semantic graph driver takes the local-table and produces the formal query using this graph.

6.2 DB DESCRIPTION

The example DB used in our system is shown in Fig 6.1. The first three are the entity relations and the last two are linking relations between these entity relations. The fields of the relations are as follows.

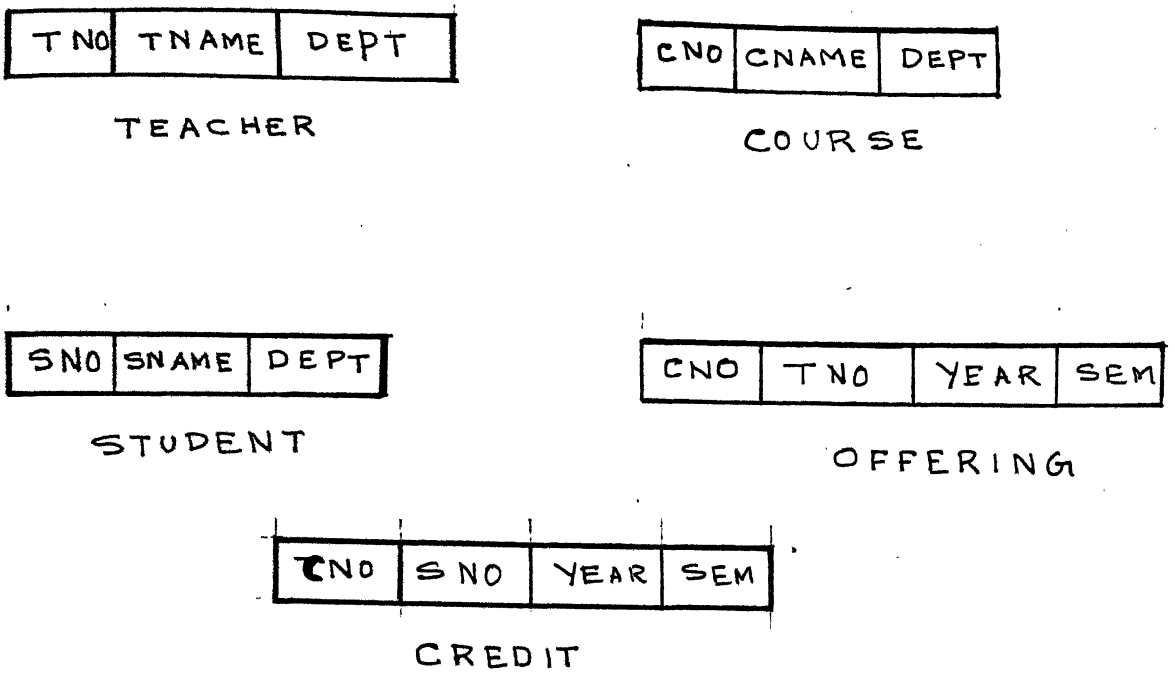


Fig 6.1. EXAMPLE DATABASE.

```
(LOCAL-TABLE (TYPE MC)
             (NEED TNO)
             (VARS (TNO (= DEPT
                           (COMPUTERSCIENCE))))
             (EXT NIL))
```

Fig 6.2. LOCAL-TABLE OF EXAMPLE
QUERY: "who is the teacher in the dept = (computer scienG)"

```
(TNO = (PROJECT TNO (SELECT
                      (= DEPT (computer
                                scienG))
                      TEACHER))
```

Fig 6.3. FORMAL QUERY OF Fig 6.2.

Teacher::Teacher number (key),teacher name, teacher dept

Course :: course number (key), course name, dept offering the course

Student :: student number (key), student name, student department

Offer :: course number, Teacher number(of the one who offers the course),

year and semester of the offering.

Credit:: course number, the student taking the course, year and sem

of the creditins.

6.3 FORMAL QUERY SYNTAX

The formal query is represented in relational algebra. We illustrate with an example.

Consider

"who is the teacher of dept = (computer science)"

The local-tabl^Ae structure of this query is shown in Fi 6.2 (a). The formal query is given in 6.2 (b). We see that the syntax is almost the same as relational algebra. The formalism is

"(<fname> = <relational algebra expression for fname>)"

where <fname> is the field needed by a clause. If a clause of the query has external field, then that field is kept as "ext" in that clause, but the relational algebra expression to get that external field precedes this clause. Similarly, if more than one clause together determine a field, then the relational algebra expression to get such a field is given as an "intersection" of these clauses. Fig 6.3 illustrates this. The second and third clauses of Fig 6.3(a) together determine a field("cno") and this field is external to the first clause. The formal query is represented as shown in Fig 6.3(c). Fig 6.3(b) shows the local-tables after dereferencing and descoping.

6.4 GRAPH DESIGN

The graph driver program takes two fields, called candidate fields, of which one is known and the other is required. It issues

" who is the teacher of the course
which is offered in the dept of Computer Science
and which is Taken by Kumar".

Fig 6.3(a) AN EXAMPLE QUERY

```
( (INTERSECT
      (LOCAL-TABLE (TYPE SC)
                   (NEED CNO)
                   (VARS (CNO (= DEPT
                                 (computer
                                    scienG))))

                   (EXT NIL) )

      (LOCAL-TABLE (TYPE COMP-CL)
                   (NEED CNO)
                   (VARS (CNO (= SNAME
                                 KUMAR)))
                   (EXT NIL) )

  )
  (LOCAL-TABLE (TYPE MC)
               (NEED TNO)
               (VARS (TNO (= CNO EXT)))
               (EXT CNO) )

)
```

Fig 6.3(b). LOCAL-TABLES OF Fig 6.3(a).

```
(CNO = (INTERSECTION
        (PROJECT CNO
                 (SELECT (= DEPT
                            (COMPUTER
                             SCIENCE))
                         COURSE)
        (PROJECT CNO
                 (JOIN SNO
                       (PROJECT SNO (= SNAME KUMAR)
                                    STUDENT)
                       (CREDIT)
                 )
        ) )

(TNO = (PROJECT TNO
               (SELECT (= CNO EXT)
                       OFFER) ) )
```

Fig 6.3 (c). FORMAL QUERY OF Fig 6.3(a).

"primitives of formal Query" which indicate what relations are to be chosen and what fields are to be joined,selected or projected, etc. By performing this primitive step iteratively, we can find the needed field from the given fields of the local-table.

The syntax of the formal query is described earlier. The design of the graph which is used to produce the formal query is discussed here.

Some times the joins which must be performed to find one candidate field from the other are obscured in English query and so, are not available in local-table. Such missing joins have to be inserted by the graph driver.

We illustrate with an example.

"who is the teacher of Cname = (data structures)".

The needed field "tno" can be found from the "cname", as follows.

Instantiate the field "cname" to "(data structures)". Select the tuples of the relation "course" which have "cname" as instantiated, and project the "cno"from these tuples. Join these with the relation "offer" on the "cno" field. Then project the "tno" fields of the tuples of offer so obtained in the previous operation.

In brief, "tno" is found from the "cname" by joining on the field "cno" into "offer" relation. This join is obscured in the query as well as in the local-table. Such insertions of the missing joins have to be done by the graph-driver process if its candidate fields so demand.

However, identification of missing joins is not unique. In the above case, we can find "tno" form "cname" by joining on the field "dept" in the relation "teacher" and then projecting "tno" field. But such an insertion of the join would mean:

"who is the teacher in the dept offering cname = (data structures)".

The underlined join is inserted in this second interpretation. But this interpretation does not mean what the query originally is intended to mean. The query assumes a join on "cno" in the "offer" relation whereas the later interpretation is making a join on "dept" field.

Thus not only an insertion of missing join, but also an appropriate choice of missing join is required in the above process.

A large potential for such an ambiguity is in prepositions and verb-phrases which connect two nouns (the two candidate fields).

For eg,

1. .... teacher of cname = ...

2. ......teacher teaching cname = ...

3. ..... teacher of dept = ...

The actual meaning of the queries 1 and 2 is

"who is the teacher of the cours which has name = (data structures)"

But such a full specfication of the join field is often hidden in NL queries. So, we see that there is a need for interpreting the path between two nouns that are connected by such concise and terse phrases.

When a user types in a query with such terse phrases he assumes that the program he is interacting with is capable of interpreting the missing join appropriately, i.e. he assumes that the program knows the semantics of the DB. He thus uses terse phrases to connect two nouns when he feels that the two nouns are very closely related to each other

and hence can be interpreted by the program. Our claim is

"a user uses a terse phrase(as in 1,2,3 above) between two nouns if he feels that the two nouns are very closely related from the DB semantics."

We now show that this close relationship between two pieces of data is used in the design of a DB. We encapsulate this knowledge into a graph and this graph is used to insert the missing Joins between the nouns connected by the terse phrases. As we pointed out earlier, the nouns of a terse phrase are closely related. Since the graph encapsulates this knowledge of closeness from the DB,the insertion of such missing Joins is appropriately done by the graph.

We proceed to show now that the DB design takes into an account this property of closeness. Actually there is a heirarchy of closeness in a DB.

The first step we do in a DB design is to decide which fragments of data are to be pieced together into a relation. We invariably keep together the attributes of a basic entity into a relation because they are closely related to each other and also they conceptually indicate a logically complete entity to the user and hence he can ask simple and

We shall illustrate the design of both the kinds of relations by an example.

In our sample data base of IITK academic database, we see that there are atleast three differnt entities

(i). courses

(ii). teachers

(iii). students.

They are three different entities and any user sees them as three conceptually differnt chunks of data. This accounts for our design of three ERs in the DB.

The next step is the design of linking relation design. We see that a teacher besides having a name and a dept also offers a course. Thus we see that the two basic entities "teacher" and "course" are related by offer. So, we keep together the keys of all the tuples of both the relations which satisfy this additional property offer. This is the reason for the linking relation offer. The remaining fields of "offer" indicate some properties of the two key fields "cno" and "tno" together.

The design of the second linking relation credit is similar.

Thus we see that there is a heirarchy in the design from the view point of closeness. We also pointed out that the nouns in a terse phrase are also very closely related in general. So, if we adopt a method which interprets the path between two nouns tersed together, by checking along this heirarchy of closeness and decide what Joins are to be inserted, the interpretation will be correct in general.

The design of our graph encapsulates this heirarchy and the missing Joins are interpreted by this graph. The design of the graph is as below.

The fields of a DB constitute the nodes of the graph. Two nodes are connected by an arc if they are Joinable or f one is the key and the other is an attribute of a relation. These arcs are undirected. Each arc is given a weight which indicates the reative closeness of the fields connected by the arc. The more the closeness the less the weight. The closest fields namely the fields of a relation are given the lowest weight. The next close arcs between keys of entity relations and the linking relations are given the next higher weight. Any other Joinable arc is given the highest weight.

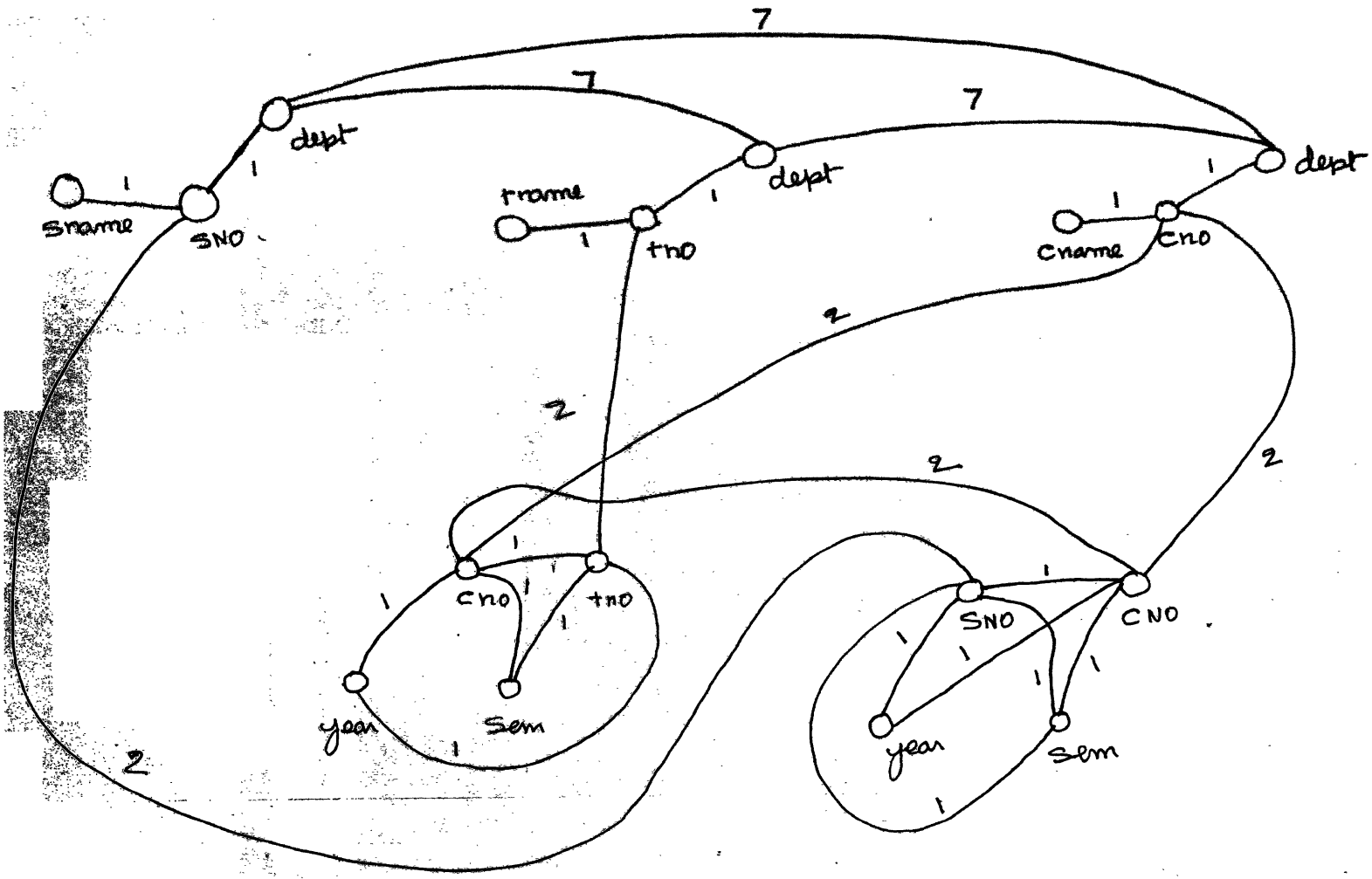


Fig 6.4. SEMANTIC GRAPH

The lowest cost path gives the most meaningful path between the fields of a terse phrase. Once the path is found, then it is very easy to identify what joins, selections, and projections are to be carried out.

The graph obtained by using the above scheme to our sample data base is shown in Fig 6.4. The graph search method to find the least weighed path is the same as the one discussed in "Principles of AI"[Nil83]. We conclude this chapter with an example.

6.5 COMPLETE EXAMPLE.

We shall illustrate the use of the graph with the following query.

"who is the teacher of sname = kumar".

The local-table is shown in Fig 6.5.

The candidate fields are "TNO" and "SNAME". We see from the Fig 6.4, that there are more than one path between these fields but, the least cost path is through the "credit" relation.

```
(LOCAL-TABLE  (TYPE  MC)
              (NEED  TNO)
              (VARS  (TNO  (= Sname KUMAR)))
              (EXT  NIL) )
```

Fig 6.5. LOCAL TABLE OF THE QUERY:
"who is the teacher of Sname = Kumar"

```
(PROJECT TNO
   (JOIN CNO
      (PROJECT CNO
         (JOIN SNO
            (PROJECT SNO (= SNAME KUMAR)
               STUDENT)
            CREDIT))
      OFFER))
```

Fig 6.6. FORMAL QUERY OF Fig 6.5.

The path is as follows. First the tuples of "student" relation which have "sname" as instantiated above are found. Then, their "sno" fields are projected. These fields are joined with the "sno" fields of the relation "credit" and the "cno" fields of the tuples of "credit" so obtained are projected. Then, these "cno" fields are joined with the relation "offer" on "cno" field. The "tno" fields of the tuples so obtained are projected. These fields provide the requeired answer.

Thus, we have interpreted the above query as follows.

"who teaches the course Which is taken by the studentw who has name = kumar".

The underlined portion sives the path inserted by the sraph. This is what is expected by the user also. The other path is alons the "dept" field of the two relations "student" and "teacher". But this interpretation does not sive the meanins intended by our example query. Thus, we see that the sraph inserts the path between two nouns of a terse phrase in an appropriate way.

The formal query of the query of Fis 6.5 is shown ir Fis 6.6.

CHAPTER 7

CONCLUSIONS

7.1 SUMMARY

Our natural language interface system has four phases, Normalisation, Parsing, Local-Table Generation, and Formal query generation.

The first phase, Normalisation, identifies the field descriptions and substitutes these with cannonical representation. Various descriptions of each field are given in terms of an ATN grammar. On recognition of each field description the actions associated with the corresponding ATN replace the description with cannonical form.

The second phase, parsing, uses an ATN parser to produce the parse of the Normalised query. The database specific information, if any, of each word is kept along with the word in the parse tree.

The third phase, Local-Table Generation, produces a query in terms of what fields are needed and what fields are specified. We call these queries skeleton queries. The database specific information required for this purpose is taken from parse structure. The heuristics of this phase use this information to produce the formal query from the

parse structure.

The fourth phase, produces the final formal query in relational algebra using the semantic graph driver. It identifies what relations have to be used, what operations like Join, select and project are to be performed. The information required by this phase is available in the Semantic Graph. This semantic graph is designed from the database for which the system works as an Interface.

The system runs on NLISP interpreter on our KL-10 processor.

7.2 CONCLUDING REMARKS

We have designed and implemented a system that provides a Natural Language Interface to a relational database.

It takes in, as parameters, the database specification and produces the inerface for the database. Although the whole process is not automated, the database specification is formalised. Translation of this database specification into system parameters is algorithmised and aided by a number of interactive functions.

The various characteristics of the database that must be used to port the system to a new database are as below

1. The descriptions of various database fields. These descriptions can conveniently be described by an ATN grammar.

2. The database specific nouns and verbs. The database specific information associated with certain nouns and verbs is used in the third phase. The information that must be provided is standardised and a set of macros are defined to put this into the lexicon.

3. The relations, their keys and the joinable fields. This information is used to construct the semantic graph. The way the semantic graph can be constructed from this information is algorithmised.

Thus, our basic objectives are achieved in designing a Natural Language Interface system that takes database specification as parameters.

7.3 FUTURE EXTENSIONS

7.3.1 ELLIPSIS

To understand the meaning of ellipsis, consider the following dialogue with a NLI system.

query1 "who taught cs415"

reply1 "karnick"

query1 "and cs625"

reply1 "malhotra"

The second query assumes the knowledge of the previous query. Such queries which do not specify the information completely and assume some information from the previous query are said to have ellipsis. Our system does not handle ellipsis. It can be extended to handle this.

*

7.3.2 EXTENSION OF THE GRAMMAR

The system presently accepts "Wh" queries only. The system can be extended to handle non "Wh" queries like yes/no queries and requesting queriess. For this, the grammar has to be extended and also the field extraction routines of local-table generation have to be changed

7.3.3 ANAPHORIC REFERENCE

If certain words of a sentence refer to what was given earlier in the sentence, then such sentences are said to have anaphoric reference. Consider

"who took cs415 from the teacher who tausht him cs325 in 83"

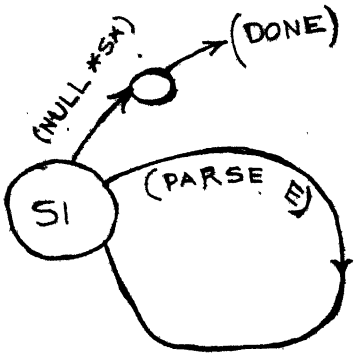
In the query above, "him" refers to "who took cs415", a part of the query. Our system does not handle such references. It can be extended to handle such references also.

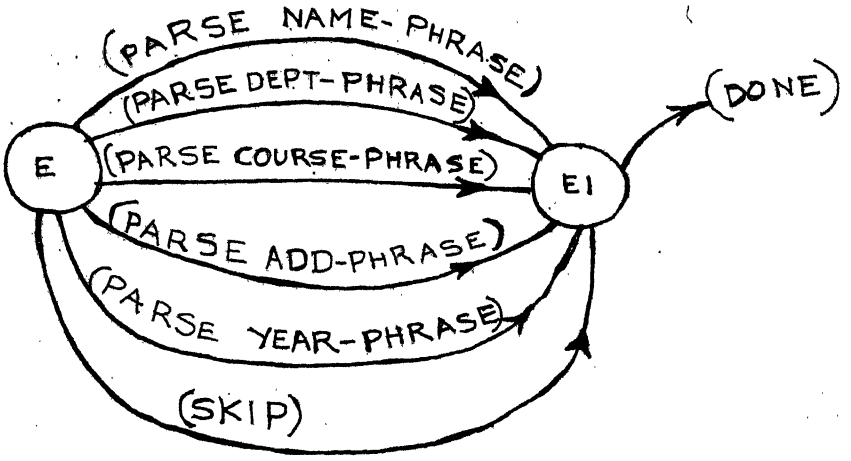
7.3.4 QUERIES ON DATABASE STRUCTURE

Queries like "what are the attributes of a student" are dependent on the database structure and are not handled by our system. It can be extended to handle such queries on database structure.

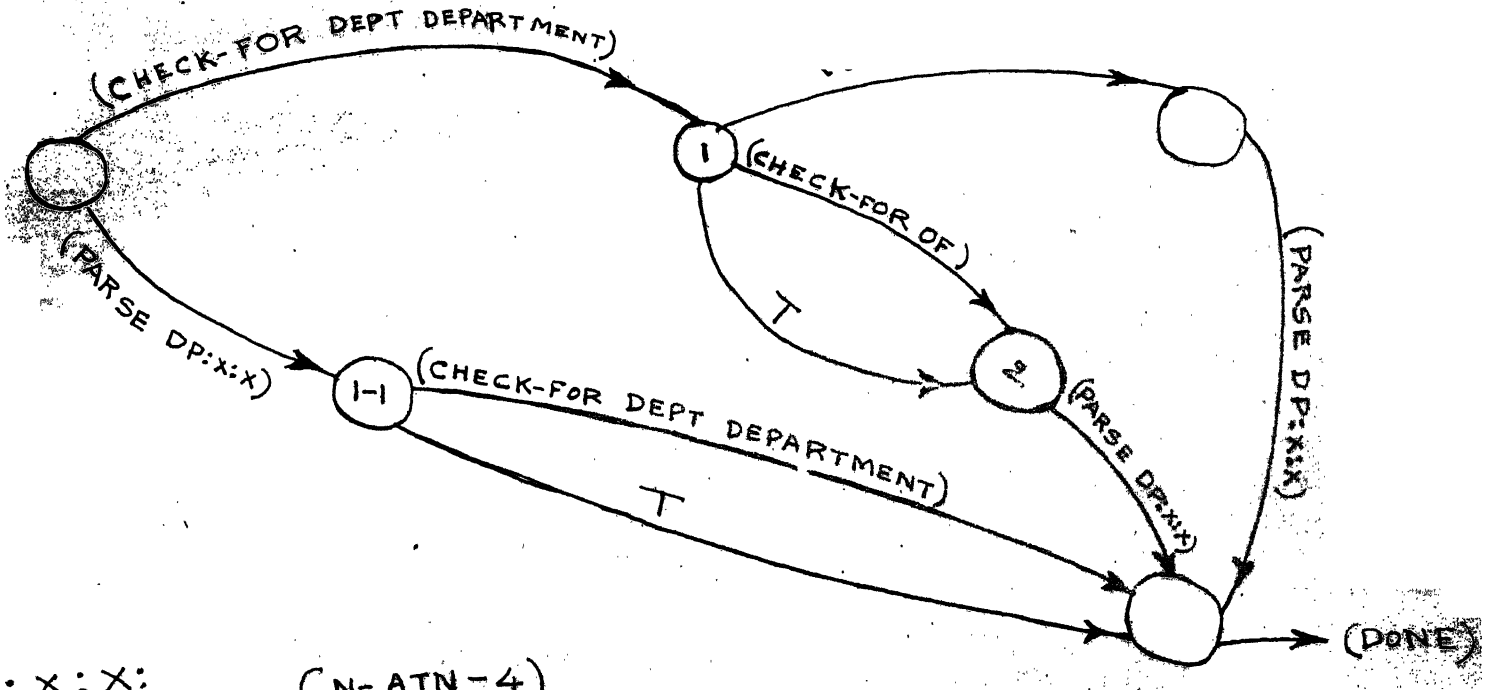
APPENDIX I

E: (N-ATN-2)

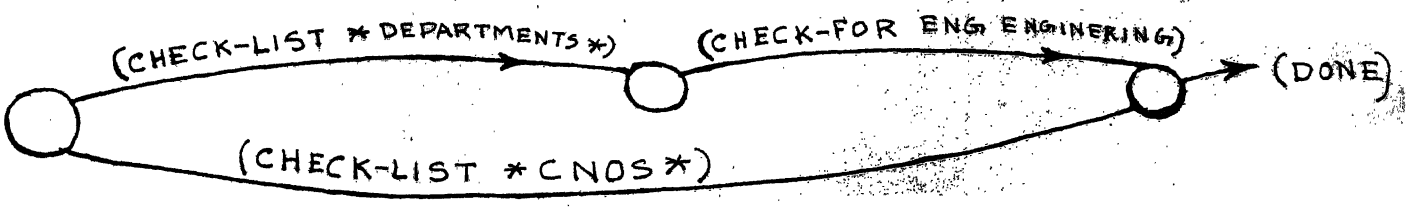




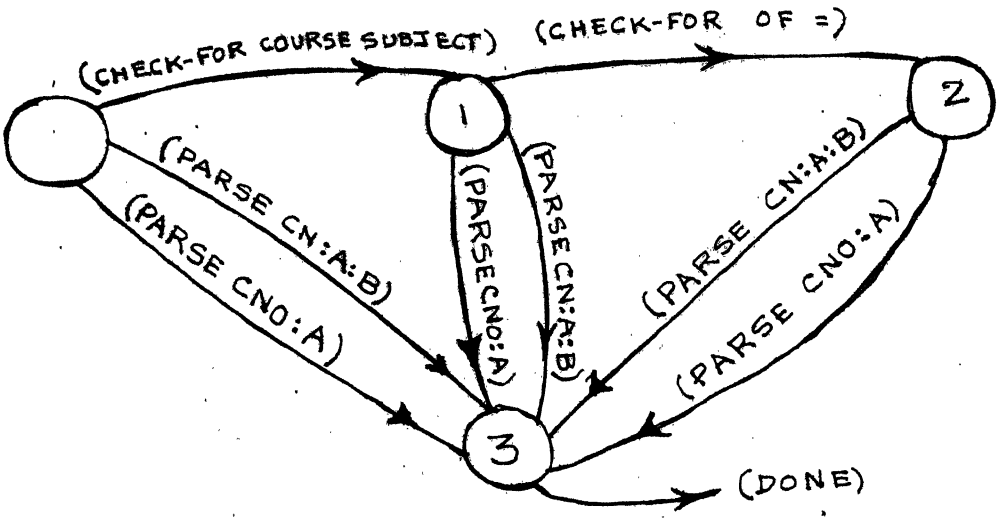
DEPT-PHRASE: (N-ATN-3)



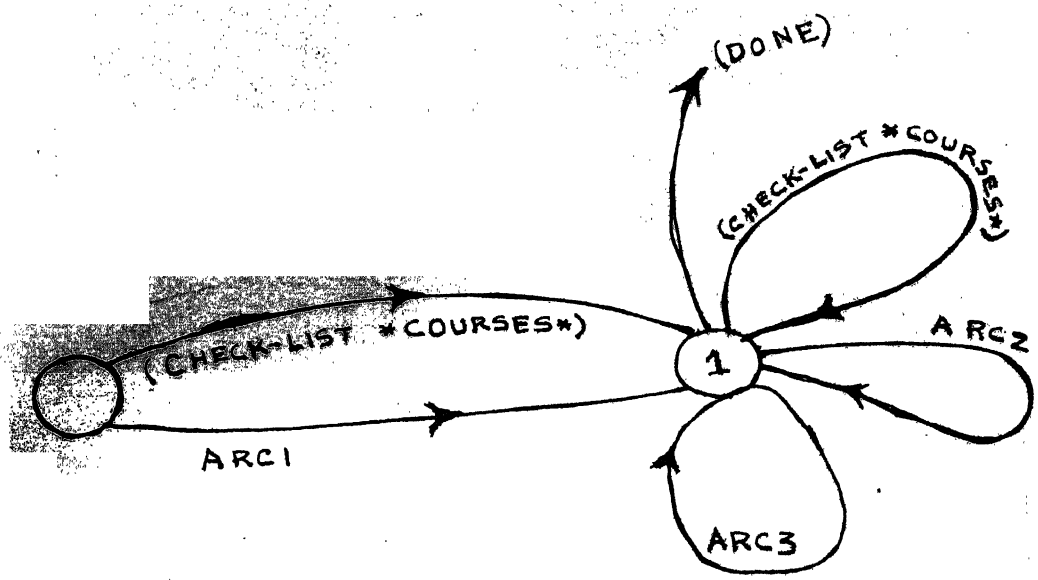
DP:X:X: (N-ATN-4)



COURSE-PHRASE (N-ATN-5)



CN:A:B (N-ATN-6)

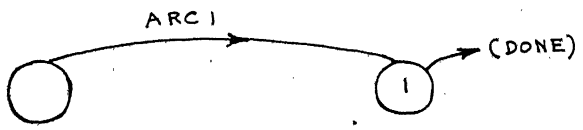


ARC 1 :: (AND [SEEK-LIST *COURSE-APPENDIX* 1] [CHECK-LIST *COURSE-APPENDIX*])

ARC2:: SAME AS ARC1

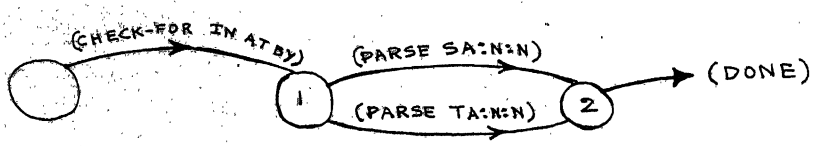
ARC 3:: (AND [OR [SEEK-LIST *COURSES* 2] [SEEK-LIST *COURSE-APPENDIX* 2]
[CHECK-FOR IN OF AND])

CNO: A (N-ATN-7)

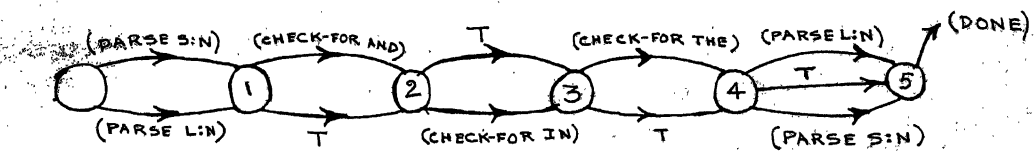


ARC1 :: (AND [> (LENGTH (EXPLODE *W*) 2)]
[NUMBERP (READLIST (CDDR (EXPLODE *W*)))]
[MEMBER (READLIST (N-CAR (EXPLODE *W*) 2)) *CNOS*])

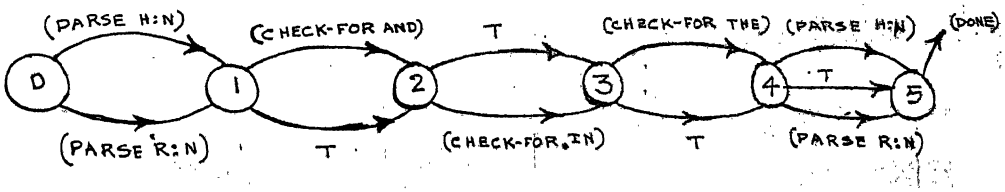
ADDRESS-PHRASE: (N-ATN-8)

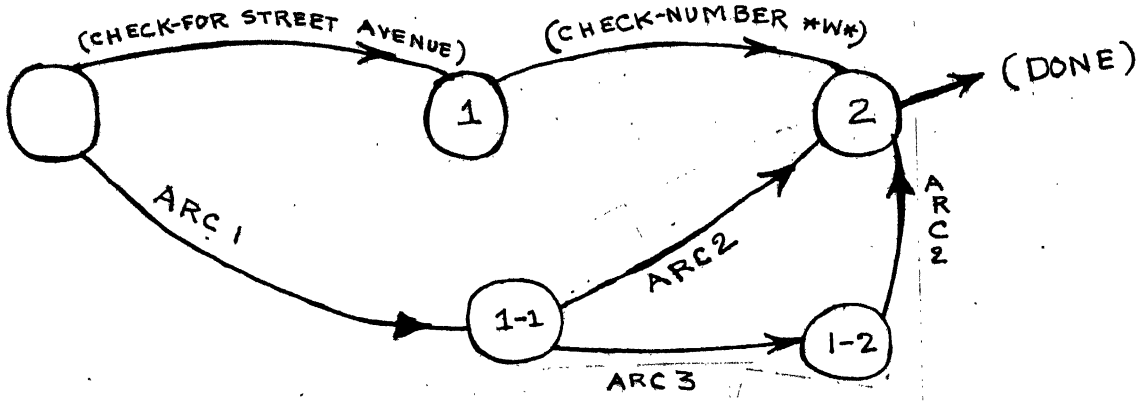


TA:N:N (N-ATN-9)



SA:N:N (N-ATN-10)



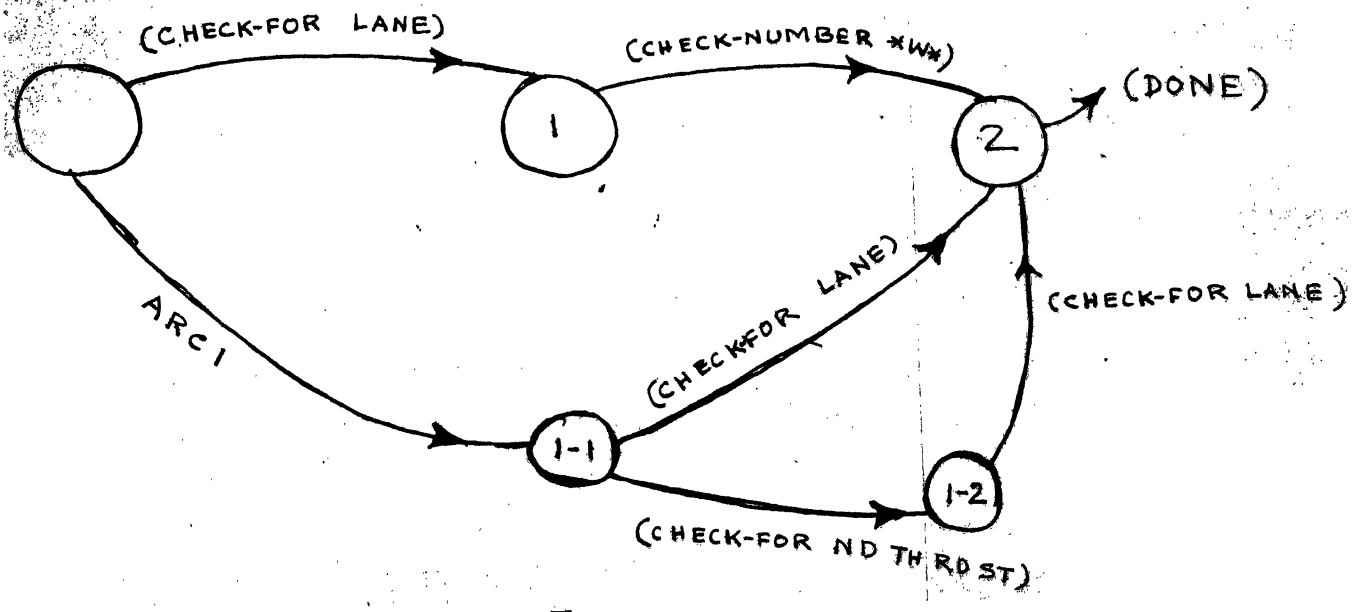


ARC1 :: (AND [SEEK 3 STREET]
[CHECK-NUMBER *W*])

ARC2 :: (CHECK-FOR STREE AVENUE)

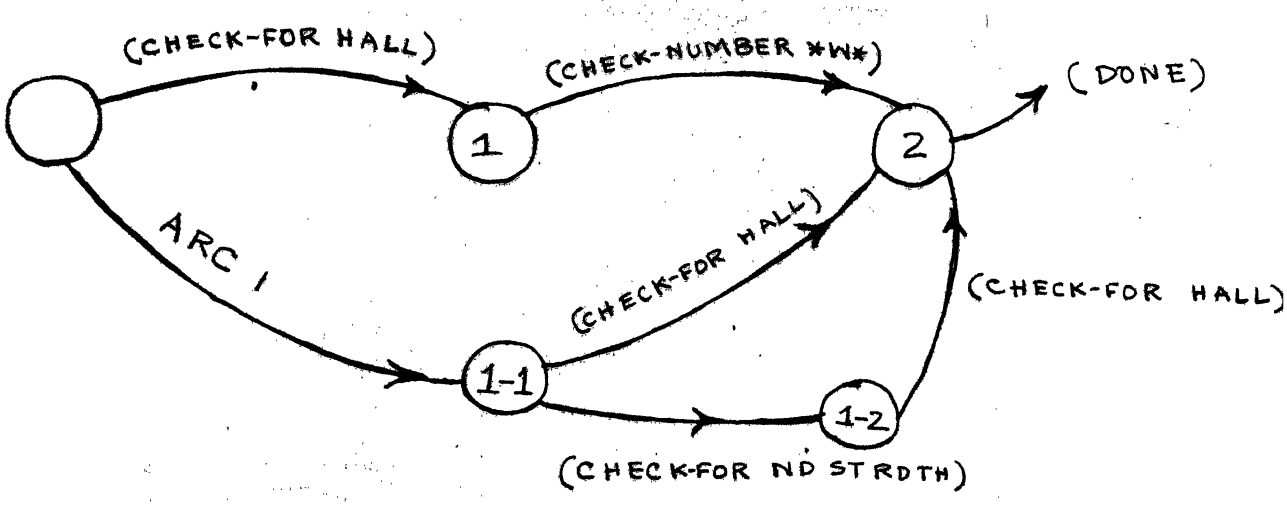
ARC3 :: (CHECK-FOR TH RD ND ST)

L:N: (N-ATN-12)



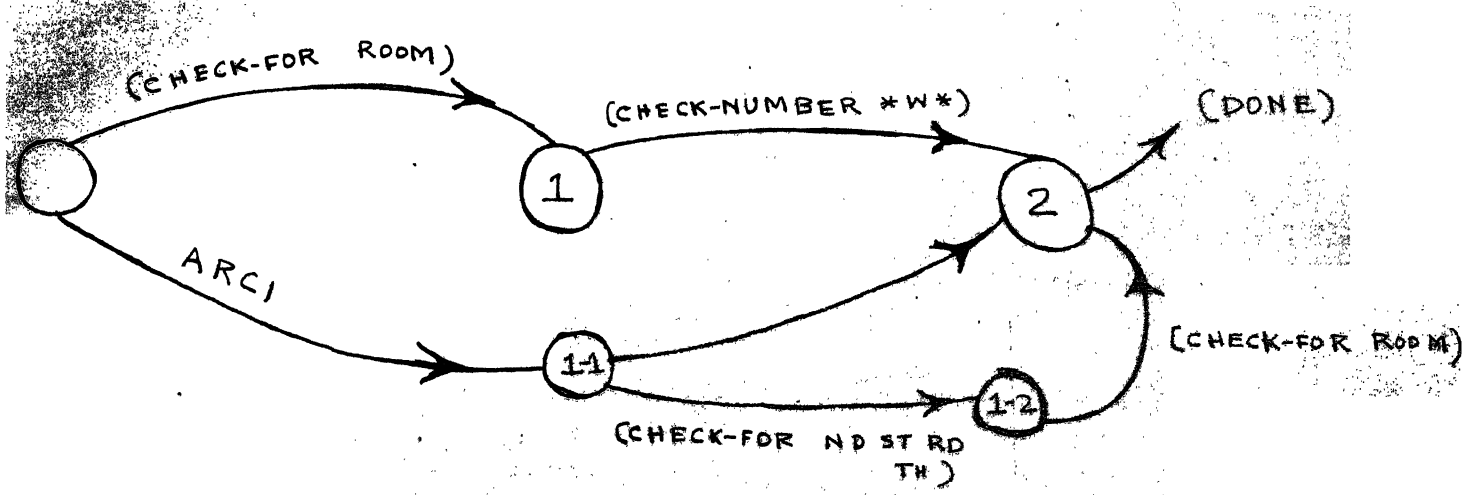
ARC1 :: (AND [SEEK 3 LANE]
[CHECK-NUMBER *W*])

:N (N-AT N-13)

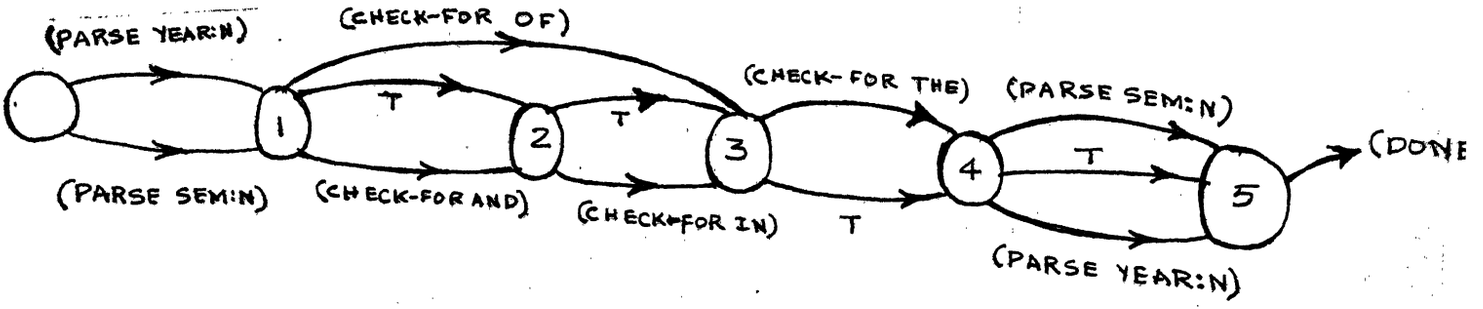


ARC1 :: (AND [SEEK 3 HALL]
[CHECK-NUMBER *W*])

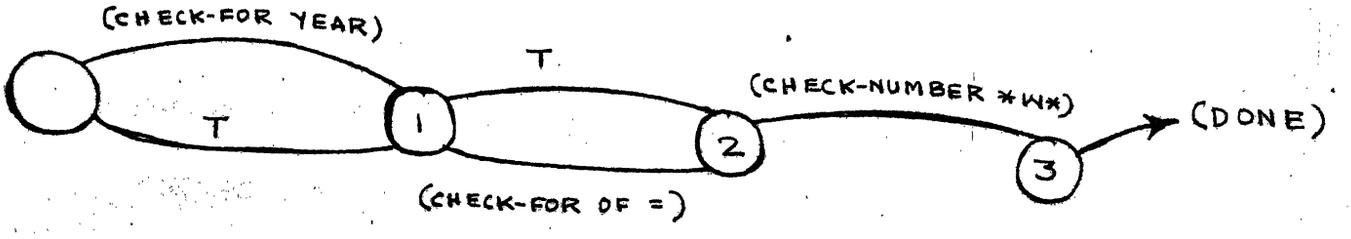
R:N (N-AT N-14)



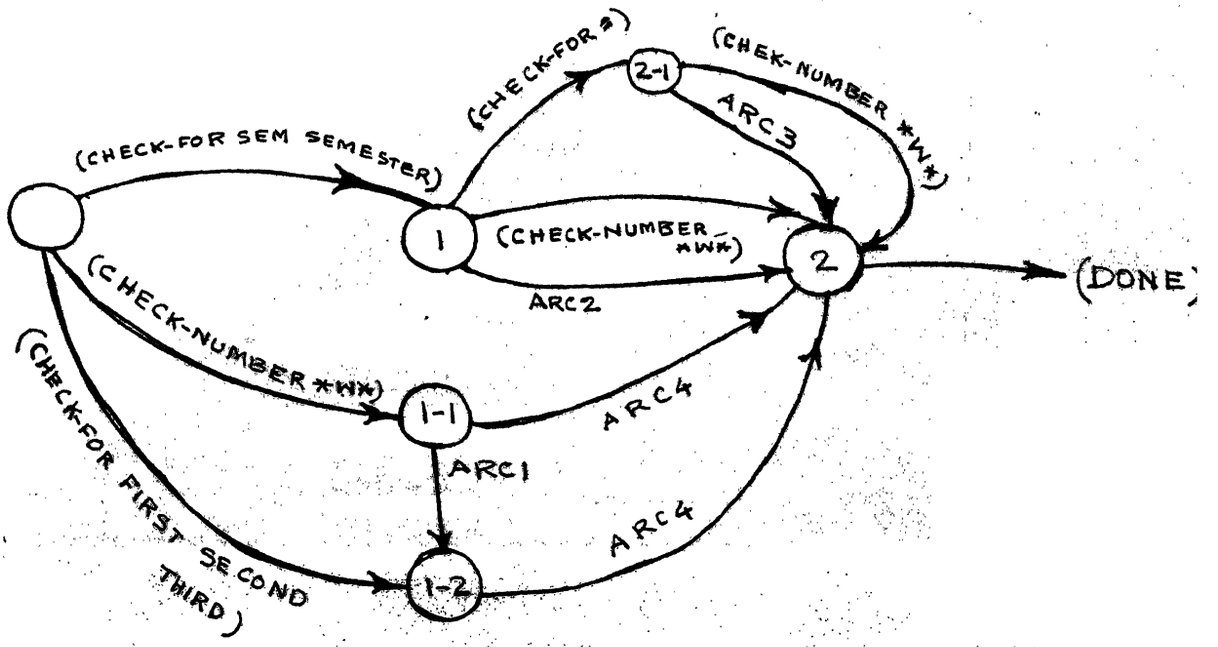
YEAR-PHRASE (N-ATN-15)



YEAR:N (N-ATN-16)



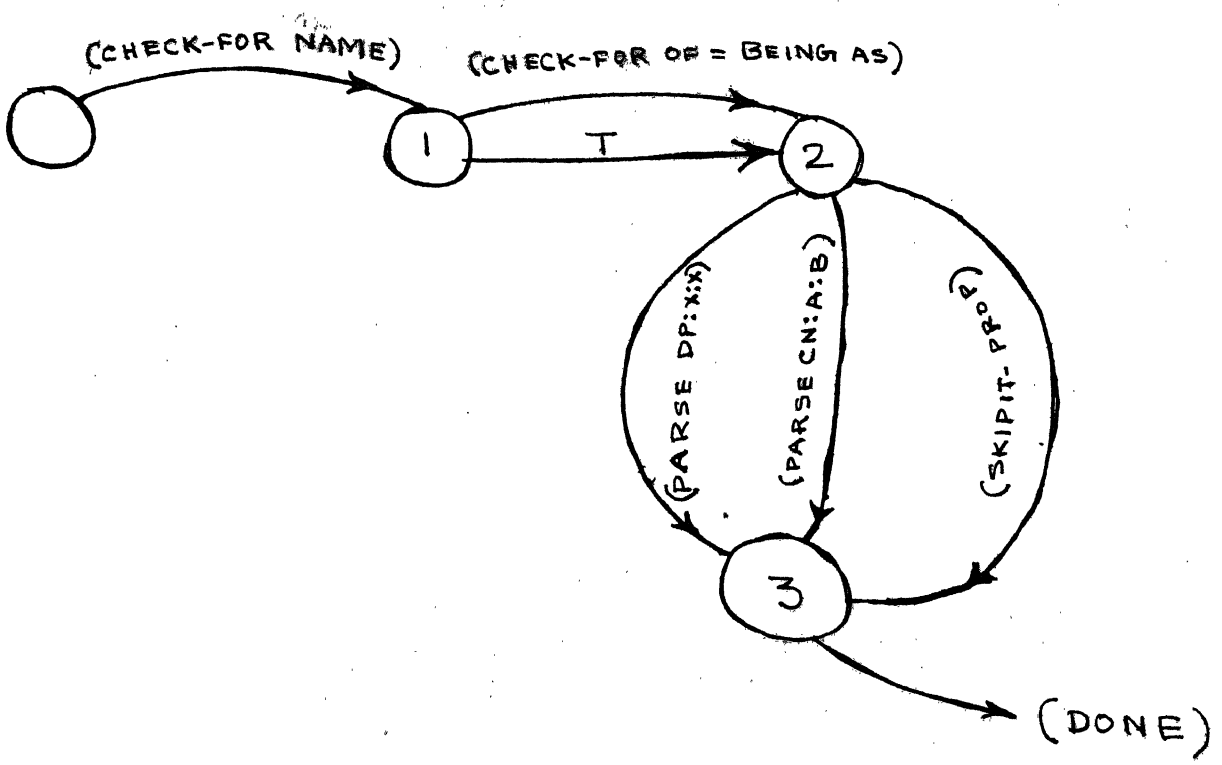
SEM:N (N-ATN-17)



ARC1 :: (CHECK-FOR ND ST RD) ; ARC3 :: (CHECK-FOR ONE TWO)

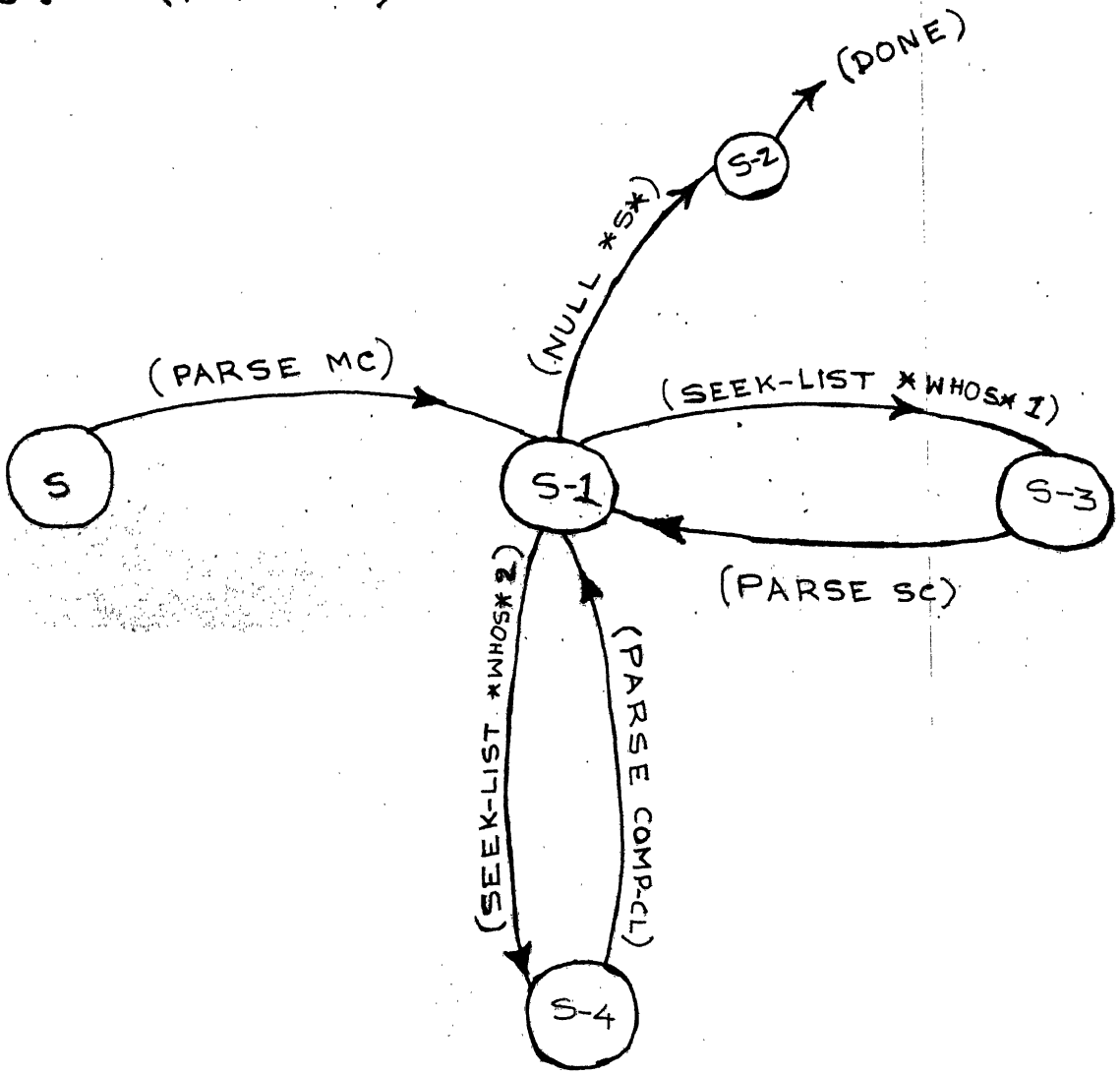
ARC2 :: SAME AS ARC3 ; ARC 4 :: (CHECK-FOR SEM SEMESTER)

NAME-PHRASE - (N-ATN-1B)

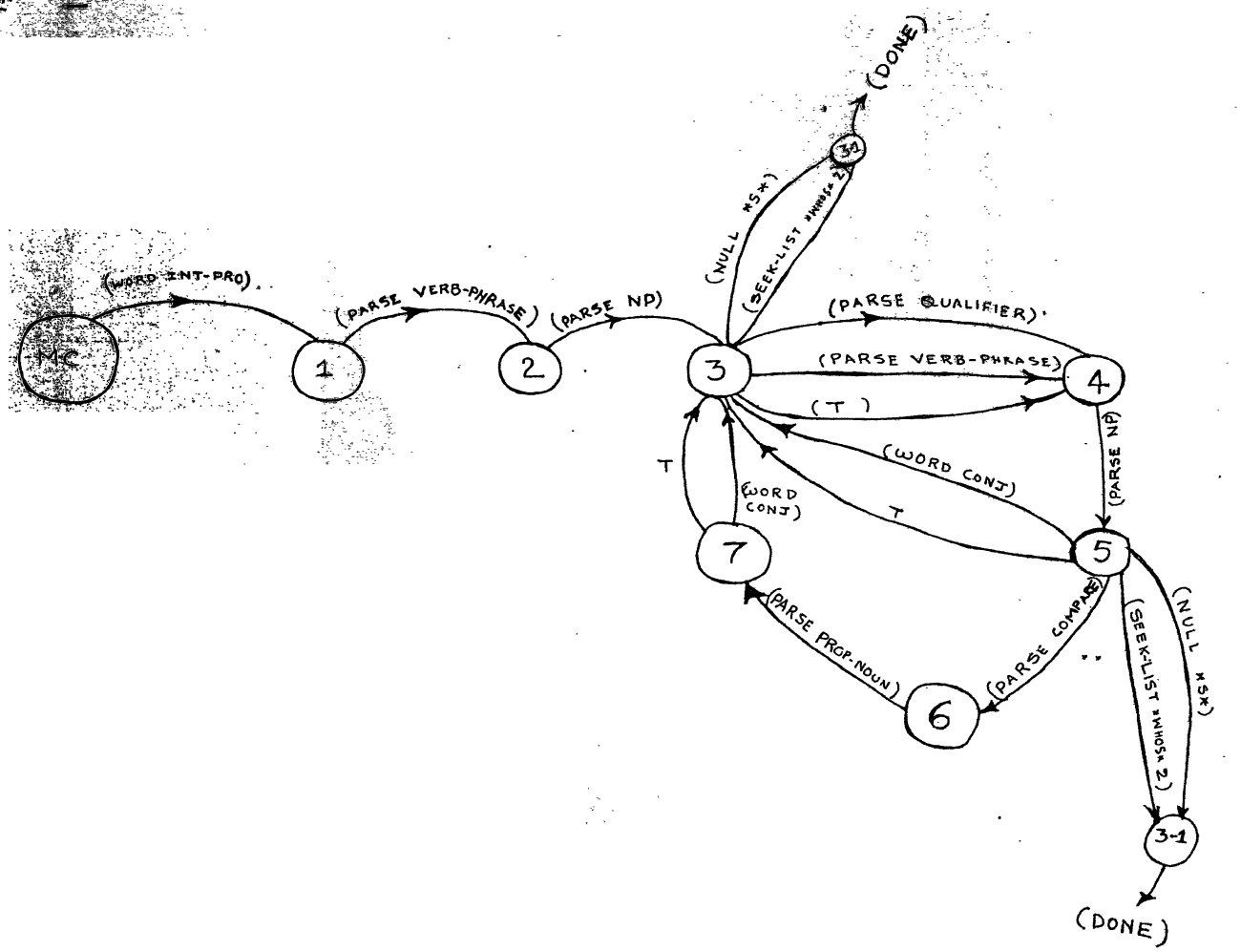


APPENDIX 2

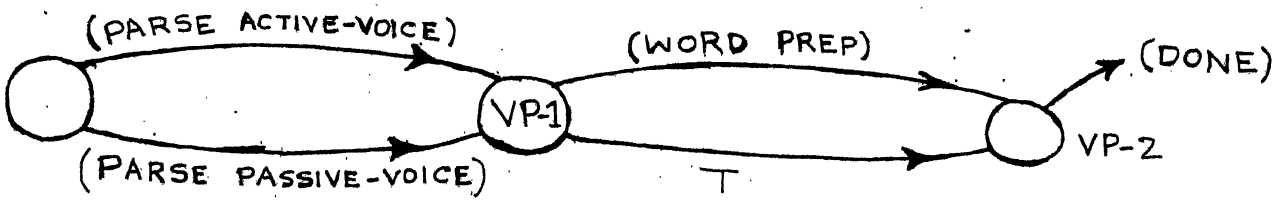
S: (P-ATN-1)


(DONE)
S-2
(NULL *S*)
(PARSE MC)
S
S-1
(SEEK-LIST *WHOS* 1)
S-3
(PARSE SC)
(SEEK-LIST *WHOS* 2)
(PARSE COMP-CL)
S-4

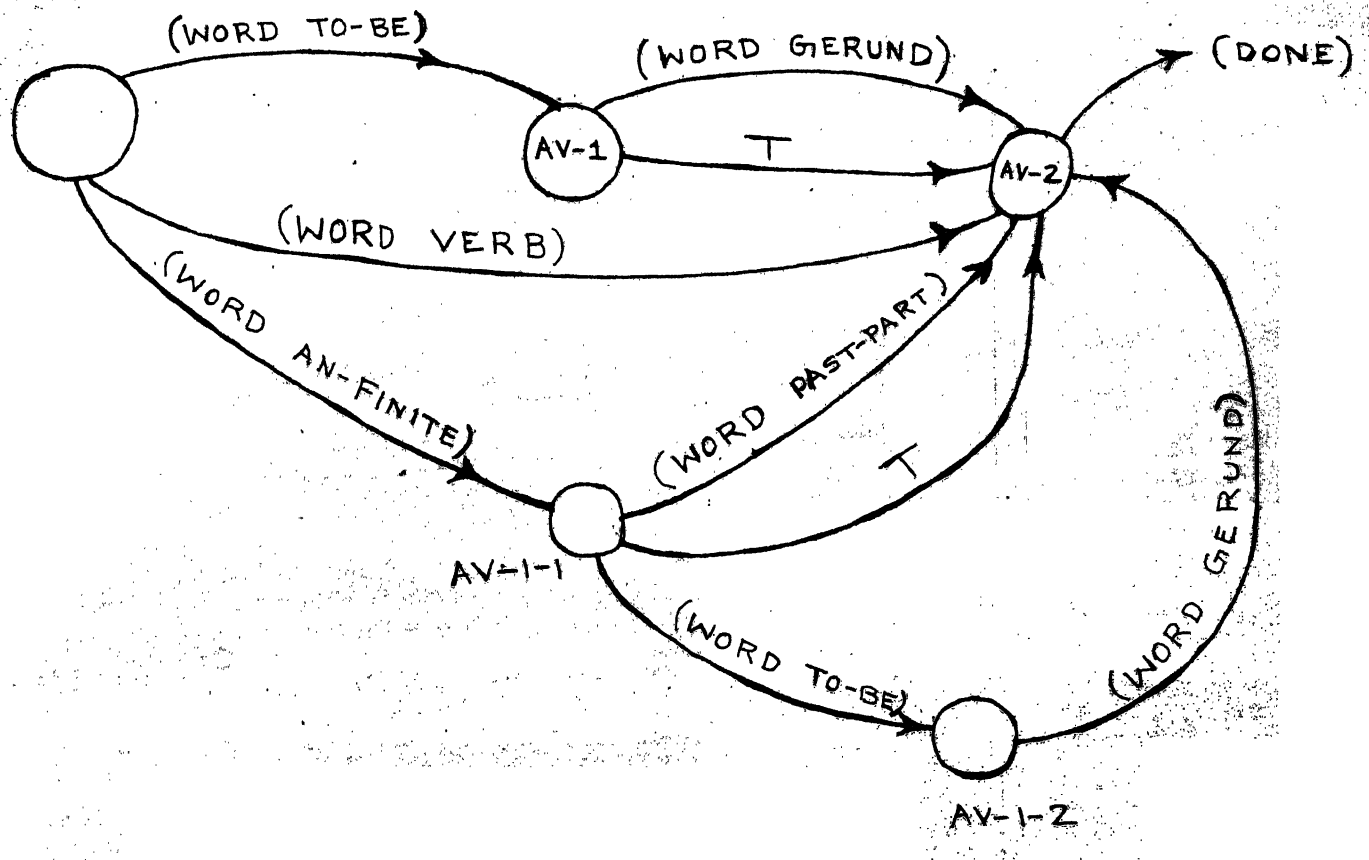
MAIN CLAUSE (P-ATN-2)



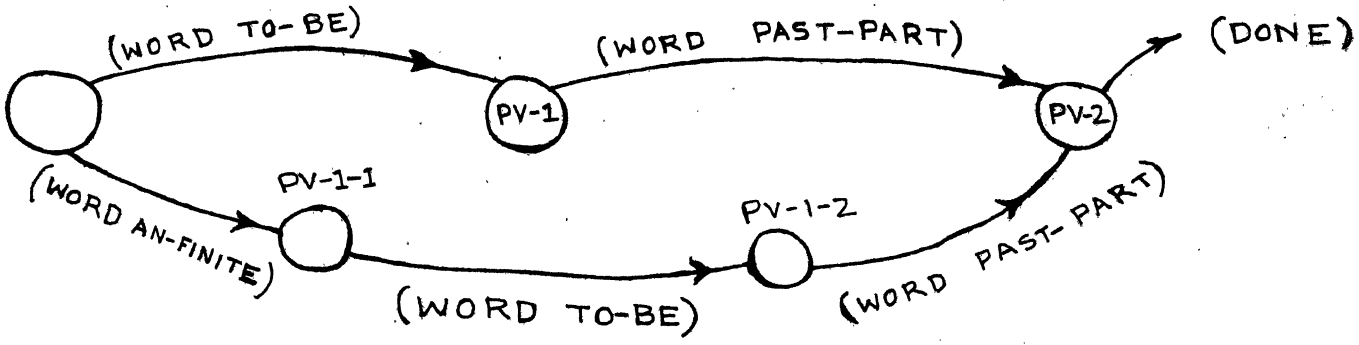
VERB-PHRASE: (P-ATN-3)



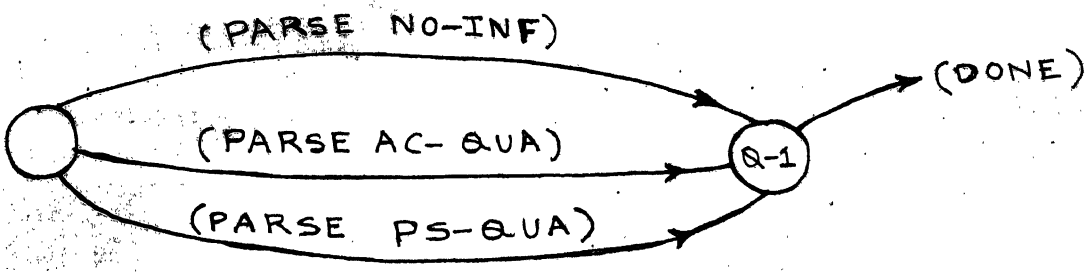
ACTIVE-VOICE: (P-ATN-4)



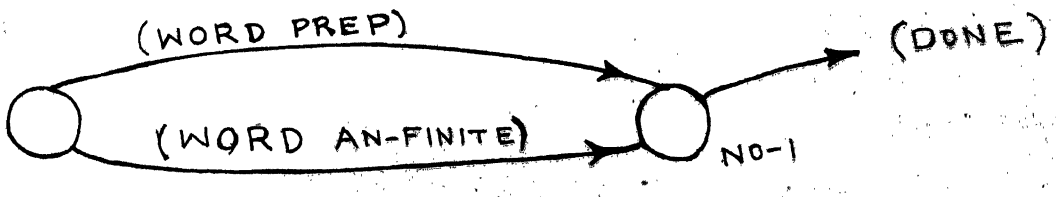
PASSIVE VOICE: (P-ATN-5)



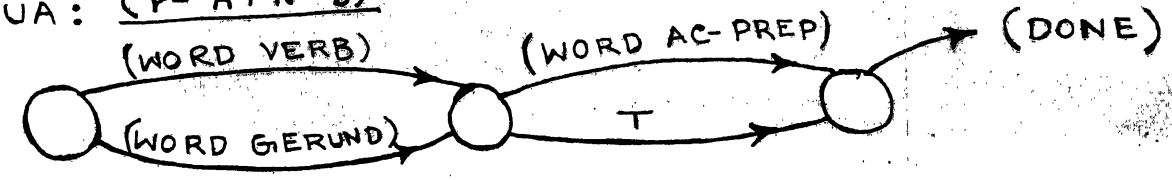
QUALIFIER: (P-ATN-6)



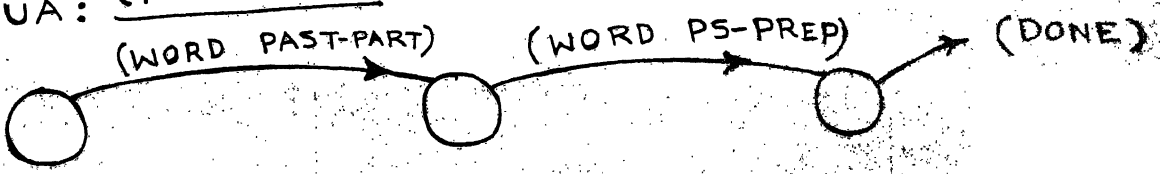
NO-INF: (P-ATN-7)



AC-QUA: (P-ATN-8)

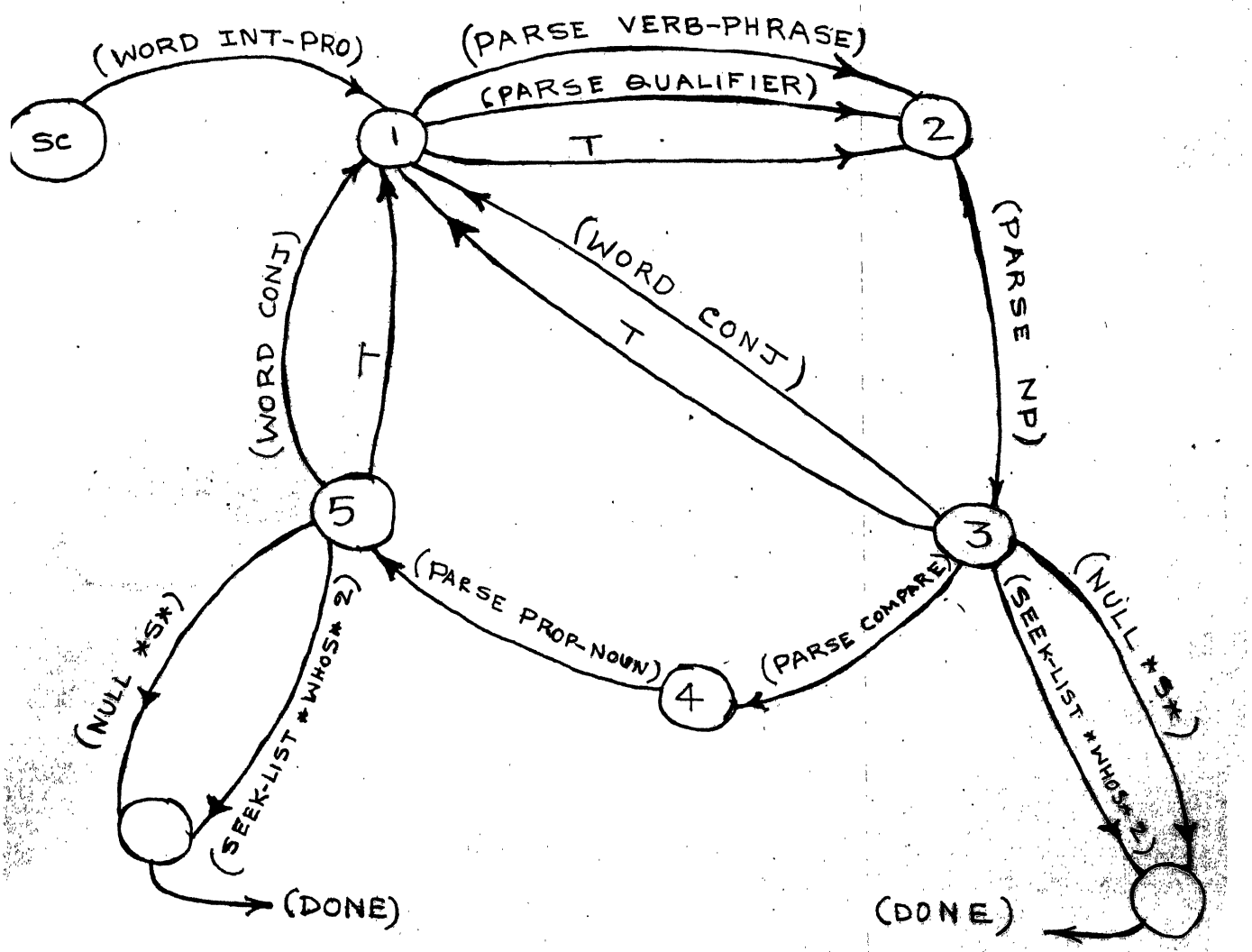


PS-QUA: (P-ATN-9)

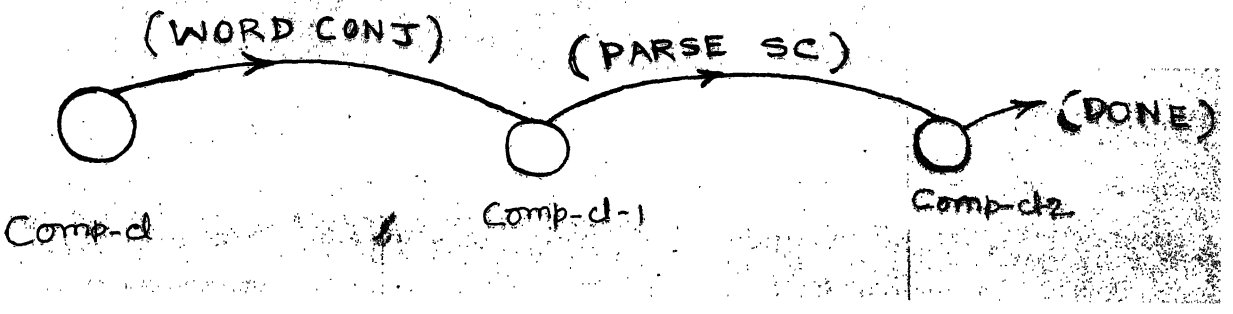


COMPARE: (P-ATN-10)

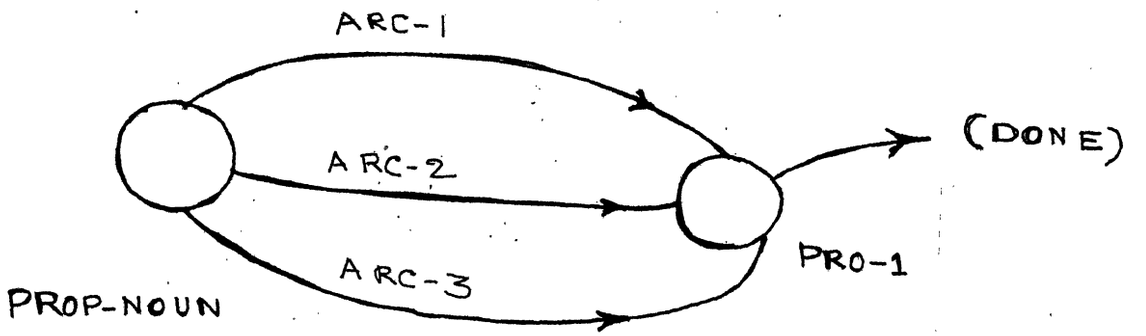
SUBORDINATE CLAUSE : (P-ATN-11)



COMPOUND CLAUSE : (P-ATN-12)



PROPER NOUN (P-ATN-13)




```
ARC-1 : ((AND [MEMBER ': (EXPLODE *W*)]
            [PUSH (APPEND (POP PARSE-STACK)
                          '((FIELD VALUE))
                          (LIST (POP *S*)))
                  PARSE-STACK]) PRO-1)

ARC-2 : ((AND [MEMBER ' PROP-NOUN
                      (GET *W* 'FEATURES)]
            [PUSH (APPEND (POP PARSE-STACK)
                          '((FIELD VALUE))
                          (LIST (POP *S*)))
                  PARSE-STACK]) PRO-1)

ARC-3: ((AND *W* [EQ (GET *W* 'FEATURES) NIL]
             [NOT (TTYMSG "Is" 2 *W* "a proper noun?")]
             [TTYESNO]
             [PUTPROP *W* '(PROP-NOUN) 'FEATURES]
             [PUSH (APPEND (POP PARSE-STACK)
                           '((FIELD NAME))
                           (LIST (POP *S*)))
                   PARSE-STACK]) PRO-1)
```

R E F E R E N C E S

1. Bolc, L. "Design of Interpreters, Compilers and Editors for ATNs", Springer Verlag, 1984.

2. Chang, C. L. "Finding Missing Joins for Incomplete Queries on Relational Databases", IBM research report, RJ2145, San Jose, California, 1978.

3. Codd, E. F. , et al. "RENDEZVOUS VERSION 1", IBM research report, RJ2144, San Jose, California, 1978.

4. Finin, T. W. "Casting RENDEZVOUS Analyser Rules into ATN Form", IBM research report, RJ2144, SanJose, California, 1978.

5. Halliday, M. A. K. , et al. "Cohesion in English", Longman Group Ltd. , London, 1976.

6. Hendrix, G. "Human Engineering for Applied Natural Language Processing", AI Centre, Stanford Research Institute, Menlo Park, California 1977.

7. Nilsson, N. J. "Principles of Artificial Intelligence", Springer Verlag, 1983.

8. Rich, E. "Natural Language Interfaces", Computer, September 1984.

9. Sacerdoti, D. "LADDER", AI centre, Stanford Research Institute, 1977.

10. Tennant, Harry "Natural Language Processing" Petrocelli Books, Inc. , New York, 1981.

11. Waltz, D. L. "An English Language Question Answering System for a Large Relational Database", Communications of the ACM, July, 1978.

12. Winograd, Terry "Language as a Cognitive Process", Volume 1, Addison-Wesley Publishing Company, Reading, Mass. , 1983.